R F BENNION

inmos

ANSI C toolset reference manual

INMOS Limited

72 TDS 225 00

August 1990

The C compiler implementation was developed from the Perihelion Software "C" Compiler and the Codemist Norcroft "C" Compiler.

INMOS document number: 72 TDS 225 00

Contents overview

Contents

Preface

About this Manual

This manual contains information about the Runtime Library functions and the implementation of ANSI C.

The manual is divided into two main parts plus appendices: The two main parts are as follows:

1 **Runtime Library**. Details of the INMOS C runtime library including summaries of all the header files and reference information about each of the library functions listed in alphabetical order.

2 **Language Reference.** Reference material for the C language and its implementation in the ANSI C toolset. Contains a summary of the new features in the ANSI standard, details about language extensions, and implementation data.

The **Appendices** describe the syntax of the language extensions and furnish ANSI compliance data.

Host versions

The manual is designed to cover the following products which represent different host versions of the toolset:

D7214	–	IBM and NEC PC running MS-DOS.
D5214	–	Sun 3 systems running SunOS
D4214	–	Sun 4 systems running SunOS
D6214	–	VAX systems running VMS

Documentation conventions

The following typographical conventions are used in this manual:

Bold type	Used to emphasize new or special terminology.
`Teletype`	Used to distinguish command line examples, code fragments, and program listings from normal text.
Italic type	In command syntax definitions, used to stand for an argument of a particular type. Used within text for emphasis and for book titles.
Braces { }	Used to denote an optional items in command syntax.
Brackets [ ]	Used in command syntax to denote optional items on the command line.
Ellipsis ...	In general terms, used to denote the continuation of a series. For example, in syntax definitions denotes a list of one or more items.
|	In command syntax, separates two mutually exclusive alternatives.

Runtime Library

72 TDS 225 00

August 1990

1 Introduction and Runtime Library summary

This chapter introduces the ANSI C Runtime Library. It describes the library header files that contain the function declarations, explains how to use them, and lists the contents of each file.

1.1 Introduction

The ANSI C Runtime Library is a library of predefined functions which perform common programming operations such as file i/o and mathematical transformations. The library supplied with the toolset is a full ANSI standard library with additional support for parallel processing, channel communication, and semaphore handling. Some additional non-ANSI functions are also provided, including `float` versions of common maths functions, low level file handling functions, and a variety of miscellaneous operations.

Library functions are declared in a number of *header files* which contain functions that are closely related with their supporting constants. The grouping of functions into a number of files makes their declaration in a program easier and ensures the correct format for declarations.

1.1.1 Reduced library

A reduced form of the library is provided for programs which do not require communication with the host system, for example, programs that run independently within embedded systems and processes on nodes which communicate only with other nodes on a transputer network.

The reduced library omits all those functions which require interaction with the server. All other functions are present, including concurrency and most non-ANSI functions. Any program that does not call any of the i/o functions or functions which depend on them, can be linked with the reduced library. Programs linked with the reduced library cannot be bootstrapped by the collector and must be configured onto a transputer network.

Three string formatting functions from the standard i/o library are separately declared in the header file `stdiored.h`, to allow them to be used in programs linked with the reduced library. Further details can be found in section 1.3.13.

Note: Programs linked with the reduced library must be collected from a configuration binary file, that is, the programs must be *configured*.

1.1.2 Accessing library functions

Library functions must be declared like any other C function, and is simply performed by including the appropriate header file; the correct file to include can be determined from the function synopsis (see chapter 2). By using the header file the co-declaration of the correct constants and macros for the function is assured.

1.1.3 Linking libraries with programs

Function code is incorporated with the program by linking in the appropriate library file.

The runtime library functions are provided in two main object files, `libc.lib` for the full library, `libcred.lib` for the reduced library, which must be linked with any application program that uses them. The file `centry.lib` is also provided for linking with programs written in mixtures of languages.

The file `collc.lib` is also supplied to support the entry points used by the earlier 3L Parallel C toolset. This system is described in appendix G '*3L functions supported*' of the accompanying User Manual.

The full and reduced libraries contain function code compiled for different transputer types and error modes. The correct code for the transputer target and program error mode is selected at link time.

Two link startup files are provided for single and multitransputer C programs which use the full library (`startup.lnk`) and for multitransputer C programs which use the reduced library (`startrd.lnk`). The startup files contain commands to direct the linker to select code from the correct library file, and the libraries do not need to be specified on the linker command line.

Library files are indexed to assist module selection by the linker.

1.1.4 ISERVER protocols

All functions in the library use the communication protocols of the the host file server to perform program i/o. These protocols are invisible to the C applications programmer. ISERVER protocol and its underlying functions are described in appendix D '*ISERVER protocol*' of the accompanying User Manual.

The library function **server_transaction** provides access to low level IS-ERVER functions.

1.1.5 Functions which require static

Certain functions in the Runtime Library require static values. If these functions are called simultaneously by two concurrent processes there may be contention for the same data area and return values may be unpredictable.

Functions which should be used with great care in concurrently executing processes are as follows:

asctime	getenv	localtime	rand	set_abort_action
signal	stdlib	strerror	strtok	tmpnam

More information about the the use of these functions can be found under the detailed function descriptions in chapter 2.

The global variable **errno** should also be used with great care in a concurrent environment since there is no protection on its assignment.

1.2 Header files

Header files contain functions declarations, macros, and other definitions grouped together for convenient reference in a program. Header files generally contain declarations of related functions along with definitions of supporting constants and other declarations. Header files may consist only of macros and constant, for example, **limits.h**.

Header files supplied with the ANSI C toolset are listed in Table 1.1.

The rest of this chapter describes the contents of the header files and is divided into three sections covering the three main groups of files: ANSI standard functions; Concurrency functions; and Other functions. Header files in each main group are described under generic subheadings, **<stdio.h>** is described under the heading "Standard i/o".

Header file	Description
`assert.h`	Diagnostics.
`channel.h`	Channel handling.
`ctype.h`†	Character handling and manipulation.
`dos.h`	DOS specific operations.
`errno.h`†	Error handling.
`float.h`†	Real number arithmetic.
`host.h`	Host system information.
`iocntrl.h`	Low level file handling.
`limits.h`†	Language implementation limits.
`locale.h`†	Locale specific data.
`math.h`†	Maths and trig functions.
`mathf.h`	`float` versions of maths and trig functions.
`misc.h`	Miscellaneous functions.
`process.h`	Process startup, handling, and control.
`semaphor.h`	Semaphore handling.
`setjmp.h`†	Non-local jumps.
`signal.h`†	Signal handling.
`stdarg.h`†	Variable argument handling.
`stddef.h`†	Standard definitions.
`stdio.h`†	Standard i/o and file handling.
`stdiored.h`†	Reduced library string formatting functions.
`stdlib.h`†	General programming utilities.
`string.h`†	String handling and manipulation.
`time.h`†	System clock date and time.
†ANSI standard files	

Table 1.1 ANSI C toolset header files

1.3 ANSI functions

ANSI functions are contained in a series of header files defined in the ANSI standard. They encompass standard function sets such as file i/o, maths and trig functions, character and string handling, error handling, and many other functions in common usage within existing non-ANSI environments.

1.3.1 Diagnostics <assert.h>

The header file `assert.h` contains a single macro definition:

Function	Description
`assert`	Inserts a diagnostic line into a program.

The definition of `assert` depends upon the value of the macro `NDEBUG`, which is not itself defined in `assert.h`.

1.3.2 Character handling <ctype.h>

The header file `ctype.h` declares a set of functions for character identification and manipulation. The file also contains character range macros, not listed here.

Function	Description
`isalnum`	Determines whether a character is alphanumeric.
`isalpha`	Determines whether a character is alphabetic.
`iscntrl`	Determines whether a character is a control character.
`isdigit`	Determines whether a character is a decimal digit.
`isgraph`	Determines whether a character is a printable non-space character.
`islower`	Determines whether a character is a lower-case letter.
`isprint`	Determines whether a character is a printable character (including space).
`ispunct`	Determines whether a character is a punctuation character.
`isspace`	Determines whether a character is one which affects spacing.
`isupper`	Determines whether a character is an upper-case letter.
`isxdigit`	Determines whether a character is a hexadecimal digit.
`tolower`	Converts an upper-case letter to its lower-case equivalent.
`toupper`	Converts a lower-case letter to its upper-case equivalent.

1.3.3 Error handling <errno.h>

The header file `errno.h` declares the error variable `errno` and defines codes for the values to which it may be set. The file also contains a number of other error codes, not listed here, which are included for compatibility with earlier INMOS compiler toolsets.

Variable	Description
`errno`	A variable of type `volatile int`. Set to a positive error codes by several library routines.

Error code	Description
`EDOM`	The argument to a floating point function is out of range.
`ERANGE`	Overflow or underflow in a floating point function.
`ESIGNUM`	Illegal signal number supplied to `signal`.
`EIO`	Error in low level i/o function used to communicate with the server.
`EFILPOS`	Error in file positioning functions `ftell`, `fgetpos`, or `fsetpos`.

1.3.4 Floating point constants <float.h>

Macro	Description
FLT_RADIX	Radix of exponent representation.
FLT_ROUNDS	Rounding mode for floating point addition.
FLT_MANT_DIG	Number of digits in a float mantissa.
DBL_MANT_DIG	double form of FLT_MANT_DIG.
LDBL_MANT_DIG	long double form of FLT_MANT_DIG.
FLT_EPSILON	Minimum number of type float such that 1.0 + x != 1.0
DBL_EPSILON	double form of FLT_EPSILON.
LDBL_EPSILON	long double form of FLT_EPSILON.
FLT_DIG	Number of decimal digits of precision for float parameters.
DBL_DIG	double form of FLT_DIG.
LDBL_DIG	long double form of FLT_DIG.
FLT_MIN_EXP	Minimum float exponent.
DBL_MIN_EXP	double form of FLT_MIN_EXP.
LDBL_MIN_EXP	long double form of FLT_MIN_EXP.
FLT_MIN	Min normalised positive number of type float.
DBL_MIN	double form of FLT_MIN.
LDBL_MIN	long double form of FLT_MIN.
FLT_MIN_10_EXP	Minimum negative integer such that 10 raised to that power is a normalised float number.
DBL_MIN_10_EXP	double form of FLT_MIN_10_EXP.
LDBL_MIN_10_EXP	long double form of FLT_MIN_10_EXP.
FLT_MAX_EXP	Max integer such that FLT-RADIX raised to that power minus 1 is a valid float number.
DBL_MAX_EXP	double form of FLT_MAX_EXP.
LDBL_MAX_EXP	long double form of FLT_MAX_EXP.
FLT_MAX	Maximum representable number of type float.
DBL_MAX	double form of FLT_MAX.
LDBL_MAX	long double form of FLT_MAX.
FLT_MAX_10_EXP	Maximum integer such that 10 raised to that power is a valid float number.
DBL_MAX_10_EXP	double form of FLT_MAX_10_EXP.
LDBL_MAX_10_EXP	long double form of FLT_MAX_10_EXP.

1.3.5 Implementation limits <limits.h>

`limits.h` defines a number of implementation constants in ANSI C.

Macro	Description
`CHAR_BIT`	The number of bits in a byte.
`SCHAR_MIN`	Min value for an object of type `signed char`.
`SCHAR_MAX`	Max value for an object of type `signed char`.
`UCHAR_MAX`	Max value for an object of type `unsigned char`.
`CHAR_MIN`	Min value for an object of type `char`.
`CHAR_MAX`	Max value for an object of type `char`.
`SHRT_MIN`	Min value for an object of type `short int`.
`SHRT_MAX`	Max value for an object of type `short int`.
`USHRT_MAX`	Max value for an object of type `unsigned short int`.
`INT_MIN`	Min value for an object of type `int`.
`INT_MAX`	Max value for an object of type `int`.
`UINT_MAX`	Max value for an object of type `unsigned int`.
`LONG_MIN`	Min value for an object of type `long int`.
`LONG_MAX`	Max value for an object of type `long int`.
`ULONG_MAX`	Max value for an object of type `unsigned long int`.
`MB_LEN_MAX`	Max number of bytes in a multibyte character.

1.3.6 Localisation <locale.h>

The header file `locale.h` defines two functions, some macros for use by `setlocale`, and a single structure.

Function	Description
`setlocale`	Sets or interrogates part of the program's locale.
`localeconv`	Assigns appropriate values to components in objects of type `struct lconv` for the formatting of numeric quantities, according to the rules of the current locale.

Macro	Description
`LC_ALL`	Names the entire locale (that is, all of the following macros).
`LC_COLLATE`	Used in the string locale functions `strcoll` and `strxfrm`.
`LC_CTYPE`	Used in the character handling functions.
`LC_NUMERIC`	Selects the decimal point.
`LC_TIME`	Used in the locale dependent time functions.
`LC_MONETARY`	Affects monetary formatting information returned by the `localeconv` function.

Structure	Description
`lconv`	A structure which describes a complete locale. Components of `lconv` are those of the standard ANSI C locale, which is the only locale supported by the ANSI C toolset.

ANSI C supports only the standard "C" locale, which has the following features:

- The execution character set comprises all 256 values 0–255. Values 0–127 represent the ASCII character set.
- The collation sequence of the execution character set is the same as for plain ASCII.
- Printing is from left to right.
- The decimal point character is '`.`'.

No other locales are permitted.

1.3.7 Mathematics library <math.h>

`math.h` declares general maths functions and their associated constants.

Note: All functions declared in `math.h` return the value 0.0 on domain errors and set `errno` to `ERANGE` on underflow errors.

Function	Description
`acos`	Calculates the arc cosine of the argument.
`asin`	Calculates the arc sine of the argument.
`atan`	Calculates the arc tangent of the argument.
`atan2`	Calculates the arc tangent of argument 1/argument 2.
`ceil`	Calculates the smallest integer which is not less than the argument.
`cos`	Calculates the cosine of the argument.
`cosh`	Calculates the hyperbolic cosine of the argument.
`exp`	Calculates the exponential of the argument.
`fabs`	Calculates the absolute value of a floating point number.
`floor`	Calculates the largest integer which is not greater than the argument.
`fmod`	Calculates the floating point remainder of argument 1/argument 2.
`frexp`	Separates a floating point number into a mantissa and an integral power of 2.
`ldexp`	Multiplies a floating point number by an integer power of 2.
`log`	Calculates the natural logarithm of the argument.
`log10`	Calculates the base 10 logarithm of the argument.
`modf`	Splits the argument into fractional and integral parts.
`pow`	Calculates x to the power y.
`sin`	Calculates the sine of the argument.
`sinh`	Calculates the hyperbolic sine of the argument.
`sqrt`	Calculates the tangent of the argument.
`tan`	Calculates the tangent of the argument.
`tanh`	Calculates the hyperbolic tangent of the argument.

Constant	Value
`HUGE_VAL`	A constant value returned if overflow or underflow occurs.

1.3.8 Non-local jumps <setjmp.h>

The header file `setjmp.h` declares two functions used to perform non-local gotos, and a single variable used by them.

Function	Description
`longjmp`	Performs a non-local jump to a given environment.
`setjmp`	Sets up a non-local jump.

The two functions are used in conjunction to first set a position (`setjmp`), then jump to this position (`longjmp`). When `longjmp` executes, it appears to the user as if the program had just returned from the call to `setjmp`. The `setjmp` must always be at a higher level than the corresponding `longjmp`.

Variable	Meaning
`jmp_buf`	An array type used to save a calling environment.

1.3.9 Signal handling <signal.h>

The header file `signal.h` defines two functions for signal handling, one type, and several constants.

Function	Description
`raise`	Forces a pseudo-exception via the signal handler.
`signal`	Defines the way in which errors and exceptions are handled.

Type	Description
`sig_atomic_t`	Defines an atomic variable. This is a variable whose state is always known, and which cannot be confused by asynchronous interrupts.

Constant	Description
SIG_DFL	Uses the default system error/exception handling for the pre-defined value.
SIG_IGN	Ignores the error/exception.
SIG_ERR	Returned when the signal handler is invoked in error.
SIGABRT	Abort error.
SIGFPE	Arithmetic exception.
SIGILL	Illegal instruction.
SIGINT	Attention request from user.
SIGSEGV	Bad memory access.
SIGSTERM	Termination request.
SIGIO	Input/output possible.
SIGURG	Urgent condition on I/O channel.
SIGPIPE	Write on pipe with no corresponding read.
SIGSYS	Bad argument to system call.
SIGALRM	Alarm clock.
SIGWINCH	Window changed.
SIGLOST	Resource lost.
SIGUSR1	User defined signal.
SIGUSR2	User defined signal.
SIGUSR3	User defined signal.

1.3.10 Variable arguments <stdarg.h>

The header file `stdarg.h` contains a three functions and a type definition. The functions are implemented as macros.

Function	Description
va_arg	Accesses a variable number of function arguments in a function definition.
va_end	Clears up after accessing variable arguments.
va_start	Initialises a pointer to a variable number of function arguments in a function definition.

Type	Description
va_list	A type used to hold information required by the variable argument functions.

1.3.11 Standard definitions <stddef.h>

The header file `stddef.h` defines a number of commonly used data types and macros.

Type	Description
`ptrdiff_t`	The signed integral type of the result of subtracting two pointers.
`size_t`	The unsigned integral type of the result of the `sizeof` operator.
`wchar_t`	An integral type whose range of values can represent distinct codes for all members of the largest extended character set amongst the supported locales.

Macro	Description
`NULL`	A null pointer constant which is returned by many library routines.
`offsetof(type, identifier)`	Expands to an integral constant expression that has type `size_t`. The value is the offset in bytes from the beginning of a structure, designated by `type` of `identifier`.

For example:

```
struct item
       {
long int x;
        long int y;
       };

offsetof(struct item, y) = 4 /* 2 for 16-bit machines */
```

1.3.12 Standard i/o <stdio.h>

The header file `stdio.h` defines the main i/o and file handling functions, three types, and several macros.

Function	Description
`clearerr`	Clears the error and end-of-file indicators for a file stream.
`fclose`	Closes a file stream.
`feof`	Tests the state of the end-of-file indicator.
`ferror`	Tests the state of the file error indicator.
`fflush`	Flushes an output stream.
`fgetc`	Reads a character from a file stream.
`fgetpos`	Gets the position of the read/write file pointer.
`fgets`	Reads a line from a file stream.
`fopen`	Opens a file.
`fprintf`	Writes a formatted string to a file.
`fputc`	Writes a character to a file stream.
`fputs`	Writes a string to a file stream.
`fread`	Reads records from a file.
`freopen`	Closes an open file, and re-opens it in a given mode.
`fscanf`	Reads formatted input from a file stream.
`fseek`	Sets the read/write file pointer to a specified offset in a file stream.
`fsetpos`	Sets the read/write file pointer to a position obtained from `fgetpos`.
`ftell`	Gives the position of the read/write pointer in the file stream.
`fwrite`	Writes records from an array into a file.
`getc`	Gets a character from a file.
`getchar`	Reads a character from standard input.
`gets`	Gets a line from standard input.
`perror`	Writes an error message to the standard error output.
`printf`	Writes a formatted string to standard output.
`putc`	Writes a character to a file stream.
`putchar`	Writes a character to standard output.
`puts`	Writes a line to standard output.
`remove`	Removes access to a file.
`rename`	Renames a file.
`rewind`	Sets the file stream's read/write position pointer to the start of the file.

Function	Description
`scanf`	Reads formatted data from standard input.
`setbuf`	Controls file buffering.
`setvbuf`	Defines the way that a file stream is buffered.
`sprintf`	Writes a formatted string to a string.
`sscanf`	Reads formatted data from a string.
`tmpfile`	Creates a temporary file.
`tmpnam`	Creates a unique filename.
`ungetc`	Pushes a character back onto a file stream.
`vfprintf`	Writes a formatted string to a file (alternative form of fprintf).
`vprintf`	Writes a formatted string to standard output (alternative form of printf).
`vsprintf`	Writes a formatted string to a string (alternative form of sprintf).

Type	Description
`size_t`	The unsigned integral type of the result of the **`sizeof`** operator.

Macro	Description
`FILE`	Defines a structure used for recording all the information that the system needs to control a file stream. The structure contains the following data: The current position in a file. A read/write error indicator. An end-of-file indicator. Information about the file buffer. A semaphore to prevent concurrent access to the file.
`fpos_t`	Defines a structure able to hold a unique specification of every position within a file.
`NULL`	A null pointer constant that is returned by many routines.

The first group of three macros in the following list define integral constants which may be used to control the action of **`setvbuf`**; the next three macros define integral constants which may be used to control the action of **`fseek`**, and the remainder in the list are used throughout the I/O library:

Macro	Description
_IOFBF	Full I/O buffering required.
_IOLBF	Line buffering required.
_IONBF	No I/O buffering required.
SEEK_SET	Start seek at start of file stream.
SEEK_CUR	Start seek at current position in file stream.
SEEK_END	Start seek at end of file stream.
BUFSIZ	The buffer size given by `setbuf`.
EOF	End of file.
L_tmpnam	The size of an array used to hold temporary file names generated by `tmpnam`.
TMP_MAX	The maximum number of unique file names generated by `tmpnam`.
FOPEN_MAX	The minimum number of files that can be open simultaneously.
FILENAME_MAX	Maximum length of filename.

Characteristics of file handling

File handling by works on *streams* and has the following features:

- File naming follows the conventions of the host system.
- Zero length files can exist if they are permitted by the host system.
- The same file can be opened multiple times. However, because there is no support for shared access within `stdoio.h` the results may be unpredictable.
- In append mode the file position indicator is initially positioned at the end of the file.
- Spaces written out to a file before the newline character are also read in.
- The last line of a text stream does not require a terminating newline character.
- A write on a text stream does not cause the associated file to be truncated beyond that point.

- No NULL characters are appended to data written to a binary stream.
- The features of file buffering are as follows:
  - In *unbuffered* streams characters appear from the source or destination as soon as possible. Transmission of characters also occurs if input is specifically requested.
  - In *line-buffered* streams a block of characters is built up and then sent to the host system when a newline character occurs. Transmission also occurs if input is specifically requested.
  - In fully buffered streams a block of characters is sent to the host system when the buffer becomes full.

  In all buffering modes characters are also transmitted if the buffer becomes full, or if the stream is explicitly flushed.

1.3.13 Reduced library i/o functions `<stdiored.h>`

The file `stdiored.h` contains declarations of three print formatting functions from `stdio.h`. They are for use in programs linked with the reduced runtime library.

Macro	Description
`sprintf`	Writes a formatted string to a string.
`sscanf`	Reads formatted data from a string.
`vsprintf`	Writes a formatted string to a string (alternative form of `sprintf`.

1.3.14 General utilities `<stdlib.h>`

The header file `stdlib.h` contains general programming utilities and associated data types, constants, and macros. Many of the functions are implemented as macros.

Function	Description
`abort`	Causes the program to abort. The abort is equivalent to an abnormal termination of the program.
`abs`	Calculates the absolute value of an integer.
`atexit`	Specifies a function to be called when the program ends.
`atof`	Converts a string of characters to a double.
`atoi`	Converts a string to an int.
`atol`	Converts a string to a long int.
`bsearch`	Searches a sorted array for a given object.
`calloc`	Allocates memory space for an array of items and initialises the space to zeros.
`div`	Calculates the quotient and remainder of a division.
`exit`	Causes normal program termination.
`free`	Frees an area of memory.
`getenv`	Searches an environment list for a matching string.
`labs`	Calculates the absolute value of a long integer.
`ldiv`	Calculates the quotient and remainder of a long division.
`malloc`	allocates a specified area of memory.
`mblen`	Determines the number of bytes in a multibyte char.
`mbtowc`	Converts a multibyte char to a code of type `wchar_t`.
`mbstowcs`	Converts a sequence of multibyte characters to a to a sequence of codes of type `wchar_t`
`qsort`	Sorts an array of objects.
`rand`	Generates a pseudo-random number.
`realloc`	Changes the size of an object in memory.
`srand`	Sets the seed for pseudo-random numbers generated by rand.
`strtod`	Converts the initial part of a string to a double and saves a pointer to the rest of the string.
`strtol`	Converts the initial part of a string to a long int and saves a pointer to the rest of the string.
`strtoul`	Converts the initial part of a string to an unsigned long int and saves a pointer to the rest of the string.
`system`	Passes a string to the host environment for execution as a host command.
`wctomb`	Converts a code of type `wchar_t` to a multibyte character.
`wcstombs`	Opposite of `mbstowcs`. Converts a sequence of codes of type `wchar_t` to a sequence of multibyte characters.

Type	Description
`size_t`	The unsigned integral type of the result of the `sizeof` operator.
`wchar_t`	An integral type whose range of values can represent distinct codes for all members of the largest extended character set amongst the supported locales.
`div_t`	The type returned by `div`.
`ldiv_t`	The type returned by `ldiv`.

Macro	Description
`NULL`	A null pointer constant which is returned by many library routines.
`EXIT_FAILURE`	An integral expression which may be used as an argument to the exit function to return unsuccessful termination status to the Host environment.
`EXIT_SUCCESS`	As `EXIT_FAILURE` but for successful termination.
`RAND_MAX`	Maximum value returned by `rand` function.
`MB_CUR_MAX`	Maximum number of bytes in a multibyte character.

1.3.15 String handling <string.h>

The header file `string.h` declares a number of string handling functions, one type, and string constants.

Function	Description
_memcpy	In line version of memcpy.
_strcpy	In line version of strcpy.
memchr	Finds the first occurrence of a character in the first n characters of an area of memory.
memcmp	Compares the first n characters of two areas of memory.
memcpy	Copies characters from one area of memory to another (no memory overlap allowed).
memmove	Copies characters from one area of memory to another (the areas can overlap).
memset	Fills a given area of memory with the same character.
strcat	Appends one string onto another.
strchr	Finds the first occurrence of a character in a string.
strcmp	Compares two strings.
strcoll	Compares two strings (transformed according to the program's locale).
strcpy	Copies one string to another.
strcspn	Counts the number of characters at the start of one string which do not match any of the characters in another string.
strerror	Converts an error number into an error message string.
strlen	Calculates the length of a string.
strncat	Appends one string onto another (up to a maximum number of characters).
strncmp	Compares the first n characters of two strings.
strncpy	Copies one string to another (up to a maximum number of characters).
strpbrk	Finds the first character in one string that is present in another string.
strrchr	Finds the last occurrence of a given character in a string.
strspn	Counts the number of characters at the start of a string which are also in another string.
strstr	Finds the first occurrence of one string in another.
strtok	Converts a string consisting of delimited tokens into a series of strings with the delimiters removed.
strxfrm	Transforms a string according to the locale and copies it into an array (up to a maximum number of characters).

Type	Description
`size_t`	The unsigned integral type of the result of the `sizeof` operator.

Macro	Description
`NULL`	A null pointer constant which is returned by many library routines.

1.3.16 Date and time <time.h>

The header file `time.h` declares a number of functions for manipulating time, four types, and some time and date constants.

In all the following functions the local time zone is defined by the host system. Daylight Saving Time is not available.

Function	Description
`asctime`	Converts the values in a `tm` structure to an ASCII string.
`clock`	Calculates the amount of processor time used.
`ctime`	Converts a calendar time to a string.
`difftime`	Calculates the difference between two calendar times.
`gmtime`	Converts a calendar time to a broken down time, expressed as coordinated universal time (UTC time). Always returns NULL, because UTC time is not available in this implementation.
`localtime`	Converts a calendar time into a `tm` structure format.
`mktime`	Converts a `tm` structure into a `time_t` value.
`strftime`	Does a formatted conversion of a tm structure to a string.
`time`	Reads the current time.

Type	Description
`size_t`	The unsigned integral type of the result of the `sizeof` operator.
`clock_t`	Used to store times in the form of ticks per second.
`time_t`	Used to store times in a fixed format.
`struct tm`	A calendar time structure.

Macro	Description
NULL	A null pointer constant which is returned by many library routines.
CLOCKS_PER_SEC	The number of clock ticks per second.

The tm structure has the following definition:

```
struct tm {

int tm_sec; /* Secs after min [0,61] */
int tm_min; /* Mins after hour [0,59] */

int tm_hour; /* Hours since midnight [0,23] */

int tm_mday; /* Day of month [1,31] */

int tm_mon; /* Months since Jan [0,11] */

int tm_year; /* Years since 1900 */

int tm_wday; /* Days since Sunday [0,6] */

int tm_yday; /* Days since Jan 1 [0,365] */

int tm_isdst; /* Daylight saving flag */
}
```

1.4 Concurrency functions

Concurrency support in the runtime library is separated into three header files: `process.h` which contains functions to set up, run, and control concurrent processes with associated constants; `channel.h` which contains functions for communicating along channels with associated channel constants such as link addresses; and `semaphor.h` which contains the semaphore support functions.

1.4.1 Process control <process.h>

Function	Description
`ProcAfter`	Delays execution of a process until after a specified time.
`ProcAlloc`	Allocates stack space and initialises a process.
`ProcAllocClean`	Frees space allocated by `ProcAlloc`.
`ProcAlt`	Causes a process to wait for a ready input from a series of channels. Channels are referenced by pointers.
`ProcAltList`	As `ProcAlt` but references an array of channel pointers.
`ProcGetPriority`	Returns the priority of the current process.
`ProcInit`	Initialises a process.
`ProcInitClean`	Frees space allocated by `ProcInit`.
`ProcPar`	Starts two or more processes in parallel.
`ProcParam`	Alters process parameters.
`ProcParList`	As `ProcPar` takes an array of processes.
`ProcPriPar`	Starts two processes in parallel, the first being executed at high priority and the second at low priority.
`ProcReschedule`	Reschedules a process, that is, places it on the end of the process queue.
`ProcRun`	Starts a process at the same priority as the calling process (the *current* priority).
`ProcRunHigh`	Starts a high priority process.
`ProcRunLow`	Starts a low priority process.
`ProcSkipAlt`	Checks specified channels for readiness to input.
`ProcSkipAltList`	As `ProcSkipAlt` but takes an array of pointers to channels.
`ProcStop`	Stops a process.
`ProcTime`	Reads the transputer clock.
`ProcTimeAfter`	Determines the sequence of two transputer clock times.
`ProcTimerAlt`	As `ProcAlt` but uses a timeout.
`ProcTimerAltList`	As `ProcAltList` but uses a timeout.
`ProcTimeMinus`	Gives the difference between two transputer clock times.
`ProcTimePlus`	Gives the result of adding two transputer clock times.
`ProcWait`	Delays execution of a process for a specified time.

Type	Description
`Process`	A structure that holds all the information about a concurrent process.

Constant	Description
`PROC_HIGH`	The value returned by `ProcGetPriority` for a high priority process.
`PROC_LOW`	The value returned by `ProcGetPriority` for a low priority process.

1.4.2 Channel communication `<channel.h>`

Function	Description
`ChanAlloc`	Allocates and initialises a channel.
`ChanIn`	Inputs a message on a channel.
`ChanInChanFail`	As `ChanIn` but incorporates the ability to reset a channel on receipt of a message sent on another channel (such as a link failure condition).
`ChanInChar`	Inputs a byte on a channel.
`ChanInit`	Initialises a channel.
`ChanInInt`	Inputs an integer on a channel.
`ChanInTimeFail`	As `ChanIn` but incorporates a timeout after which the channel is reset if no communication occurs.
`ChanOut`	Outputs a message on a channel.
`ChanOutChanFail`	As `ChanInChanFail` but for output channels.
`ChanOutChar`	Outputs a byte on a channel.
`ChanOutInt`	Outputs an integer on a channel.
`ChanOutTimeFail`	As `ChanInTimeFail` but for output channels.
`ChanReset`	Resets a channel.

Type	Description
`Channel`	The channel type.

Constant	Description
`Not_Process_P`	A special value used in process communication and scheduling. Returned by `ChanReset`.
`LINK0OUT`	Link zero output address.
`LINK1OUT`	Link one output address.
`LINK2OUT`	Link two output address.
`LINK3OUT`	Link three output address.
`LINK0IN`	Link zero input address.
`LINK1IN`	Link one input address.
`LINK2IN`	Link two input address.
`LINK3IN`	Link three input address.
`EVENT`	Event line address.

1.4.3 Semaphore handling <semaphor.h>

Function	Description
`SemInit`	Initialises a semaphore.
`SemAlloc`	Allocates and initialises a semaphore.
`SemSignal`	Releases a semaphore.
`SemWait`	Acquires a semaphore.

Type	Description
`Semaphore`	Defines a semaphore type.

Macro	Description
`SEMAPHOREINIT`	Initialises a semaphore (same action as `SemInit` but implemented as a macro.

1.5 Other functions

The header files `iocntrl.h`, `mathf.h`, `host.h`, `dos.h`, and `misc.h` contain some further extensions to the ANSI runtime library. These include UNIX-like i/o primitives; short maths functions; host system utilities; DOS specific functions; and miscellaneous functions including debugging support.

1.5.1 I/O primitives <`iocntrl.h`>

Function	Description
`close`	Low level file close.
`creat`	Low level file create.
`filesize`	Returns the size of a given file.
`getkey`	Gets the next character from the keyboard. Waits indefinitely for the next key press. Does not echo the character to the screen.
`isatty`	Checks for standard terminal streams `stdin`, `stderror` and `stdout`.
`lseek`	Low level file seek.
`open`	Low level file open.
`pollkey`	Gets the next character from the keyboard. Returns immediately if no key press is available. Does not echo the character to the screen.
`read`	Low level read-from-file.
`server_transaction`	Allows access to ISERVER functions in a controlled way.
`unlink`	Low level file remove (corresponds to ANSI standard function `remove`).
`write`	Low level write-to-file.

1.5.2 `float` maths <`mathf.h`>

The header file `mathf.h` contains declarations of the short maths functions. Short maths functions are identical to ANSI standard functions except that all arguments and results are of type `float` rather than `double`. Errors which generate the error code `HUGE_VAL` (out of range) in the ANSI functions return `HUGE_VAL_F` in the short maths functions.

Note: All functions declared in `mathf.h` return the value 0.0 on domain errors and set `errno` to `ERANGE` on underflow errors.

Function	Description
`acosf`	Calculates the arc cosine of the `float` argument.
`asinf`	Calculates the arc sine of the `float` argument.
`atanf`	Calculates the arc tangent of the `float` argument.
`atan2f`	Calculates the arc tangent of a fraction where the numerator and denominator arguments are both `floats`.
`ceilf`	Calculates the smallest integer which is not less than the `float` argument.
`cosf`	Calculates the cosine of the `float` argument.
`coshf`	Calculates the hyperbolic cosine of the `float` argument.
`expf`	Calculates the exponential function of the `float` argument.
`fabsf`	Calculates the absolute value of the `float` argument.
`floorf`	Calculates the largest integer which is not greater than the `float` argument.
`fmodf`	Calculates the floating point remainder of a fraction where the numerator and denominator arguments are both `floats`.
`frexpf`	Separates a floating point number into a mantissa and integral power of two.
`ldexpf`	Multiplies a floating point number by an integral power of two.
`logf`	Calculates the natural logarithm of the `float` argument.
`log10f`	Calculates the base-10 logarithm of the `float` argument.
`modff`	Splits the `float` argument into fractional and integral parts.
`powf`	Calculates x to the power of y where both x and y are `floats`.
`sinf`	Calculates the sine of the `float` argument.
`sinhf`	Calculates the hyperbolic sine of the `float` argument.
`sqrtf`	Calculates the square root of the `float` argument.
`tanf`	Calculates the tangent of the `float` argument.
`tanhf`	Calculates the hyperbolic tangent of the `float` argument.

1.5.3 Host utilities <host.h>

The header file `host.h` contains one function that returns host system information and a number of host system constants.

Function	Description
`host_info`	Returns information about the host system and transputer board.

Constant	Description
`_IMS_HOST_PC`	Standard PC host.
`_IMS_HOST_NEC`	NEC PC host.
`_IMS_HOST_VAX`	VAX host.
`_IMS_HOST_SUN3`	Sun 3 host.
`_IMS_HOST_SUN4`	Sun 4 host.
`_IMS_HOST_SUN386i`	Sun 386i host.
`_IMS_HOST_APOLLO`	APOLLO host.
`_IMS_OS_DOS`	DOS operating system.
`_IMS_OS_HELIOS`	HELIOS operating system.
`_IMS_OS_VMS`	VMS operating system.
`_IMS_OS_SUNOS`	SunOS operating system.
`_IMS_OS_CMS`	CMS operating system.
`_IMS_BOARD_B004`	IMS B004 PC transputer board.
`_IMS_BOARD_B008`	IMS B008 transputer module (TRAM) Motherboard.
`_IMS_BOARD_B010`	IMS B010 4-TRAM NEC PC Motherboard.
`_IMS_BOARD_B011`	IMS B011 2-TRAM VME board.
`_IMS_BOARD_B014`	IMS B014 8-TRAM VMEbus slave card.
`_IMS_BOARD_DRX11`	INMOS VAX link interface board.
`_IMS_BOARD_QT0`	Caplin QT0 VAX/VMS link interface board.
`_IMS_BOARD_B015`	IMS B015 NEC 9800 PC TRAM motherboard.
`_IMS_BOARD_CAT`	IBM CAT transputer board.
`_IMS_BOARD_B016`	IMS B016 VMEbus master/slave motherboard.
`_IMS_BOARD_UDP_LINK`	IMS UDP Link support product.

1.5.4 DOS system functions <dos.h>

The header file `dos.h` contains a number of functions for performing DOS system operations, plus one type. The file also contains definitions of associated structures, not documented here.

All the DOS specific functions return an error if they are used on operating systems other than DOS.

Function	Description
`alloc86`	Allocates a block of host memory for use with the `to86` and `from86` functions.
`bdos`	Performs a DOS function call interrupt.
`free86`	Frees a block of host memory previously allocated with `alloc86`.
`from86`	Copies a block of host memory to transputer memory.
`int86`	Raises a software interrupt. Segment registers are untouched.
`int86x`	As `int86` but also sets the processor segment registers.
`intdos`	As `int86` but specific for a DOS function call.
`intdosx`	As `intdos` but also sets the segment registers.
`segread`	Reads the segment registers.
`to86`	Copies a block of transputer memory to host memory.

Type	Description
`pcpointer`	A type that can be used to hold a standard PC pointer.

1.5.5 Miscellaneous functions <misc.h>

The header file `misc.h` declares some additional non-ANSI functions, including three debugging support functions, plus three constants that control the operation of `set_abort_action`.

Function	Description
`debug_assert`	Stops a process on a specified condition.
`debug_message`	Inserts a debugging message.
`debug_stop`	Stops a process.
`exit_repeat`	Program termination with restart. As `exit` but allows the program to be restarted on the processor.
`exit_terminate`	Terminates the server. Used for configured programs, otherwise like `exit`.
`get_param`	Reads `interface` parameters for a configured process.
`max_stack_usage`	Estimates runtime stack usage in a program.
`set_abort_action`	Sets or queries the action to be taken by `abort`. The possible actions are: exit without clearing files; or halt the transputer.

Constant	Description
`ABORT_EXIT`	Directs `set_abort_action` to cause a normal program exit on `abort`.
`ABORT_HALT`	Directs `set_abort_action` to halt the transputer on `abort`.
`ABORT_QUERY`	Directs `set_abort_action` to return the current `abort` action without resetting it.

2 Alphabetical list of functions

This chapter contains detailed reference information for the runtime library functions and their operation.

2.1 Format

Function descriptions are laid out in a standard format. First, the function name is given, highlighted in large type, followed on the same line by a brief summary of its action. A function synopsis follows which specifies the name of the header file to be included and describes the function prototype.

The function synopsis is followed by detailed information about the function under the following headings:

Heading	Information given
Synopsis:	The file to be included and the function declaration.
Arguments:	A list of the function's parameters and their meanings.
Results:	The result(s) returned.
Errors:	The action(s) taken on error.
Description:	A detailed description of the function with examples and hints on usage.
Example:	An example of the function's use, where appropriate.
See also:	A list of related functions, where appropriate.

2.1.1 Reduced library

Where functions are not available in the reduced library, this is indicated in the function description.

2.1.2 Macros

Where functions are implemented as macros, or as both macros and regular C functions, this is also indicated in the detailed description.

For these functions the version used by the compiler depends on the syntax of the calling statement. If the call uses parentheses around the function name (as in `(putchar)(ch)`), the regular function is used; if parentheses are omitted (as in `putchar(ch)`), the macro form is used instead.

2.2 List of functions

_memcpy Optimised version of memcpy.

Synopsis:

```
#include <string.h>
void *_memcpy(void *s1, const void *s2, size_t n);
```

Arguments:

`void *s1`	A pointer to the destination of the copy.
`const void *s2`	A pointer to the source of the copy.
`size_t n`	The number of characters to be copied.

Results:

Returns the unchanged value of `s1`.

Errors:

The behaviour of `_memcpy` is undefined if the source and destination overlap.

Description:

`_memcpy` copies `n` characters from the area of memory pointed to by `s2` (the source) to the area of memory pointed to by `s1` (the destination). It is identical to the ANSI defined function `memcpy` in every way except that it is compiled directly in line as transputer code if certain conditions are met. Further details can be found in section 11.4 in the accompanying User Manual.

See also:

`memcpy` `memmove`

_strcpy Optimised version of strcpy.

Synopsis:

```
#include <string.h>
char *_strcpy(char *s1, const char *s2);
```

Arguments:

`char *s1`	A pointer to the array used as the copy destination.
`const char *s2`	A pointer to the string used as the copy source.

Results:

Returns the unchanged value of `s1`.

Errors:

The behaviour of `_strcpy` is undefined if the source and destination overlap.

Description:

`_strcpy` copies the source string (pointed to by `s2`) into the destination array (pointed to by `s1`). It is identical to the ANSI defined function `strcpy` except that it is compiled directly in line as transputer code if certain conditions are met. Further details can be found in section 11.4 in the accompanying User Manual.

See also:

`strcpy strncpy`

abort Aborts the program.

Synopsis:

```
#include <stdlib.h>
void abort(void);
```

Arguments:

None.

Results:

`abort` does not return.

Errors:

None.

Description:

`abort` causes immediate termination of the program. It does not flush output streams, close open streams, or remove temporary files. `abort` passes `SIGABRT` to the signal handler, to show that the program has terminated abnormally.

The default action is to abort the program without halting the processor. The function can be set to halt the processor by first calling `set_abort_action` with the appropriate parameter.

If set to halt `abort` forces the processor to halt even if the program is not in HALT mode, by explicitly setting the Halt-On-Error and Error flags.

See also:

`set_abort_action` `exit` `exit_terminate` `signal`

`abs` Calculates the absolute value of an integer.

Synopsis:

```
#include <stdlib.h>
int abs(int j);
```

Arguments:

`int j`	An integer.

Results:

Returns the absolute value of `j`.

Errors:

If the result cannot be represented the behaviour of `abs` is undefined.

Description:

`abs` calculates the absolute value of the integer `j`.

See also:

`labs`

acos Calculates the arc cosine of the argument.

Synopsis:

```
#include <math.h>
double acos(double x);
```

Arguments:

`double x` A number in the range [-1..+1].

Results:

Returns the arc cosine of `x` in the range [0..pi] radians.

Errors:

A domain error occurs if `x` is not in the range [-1..+1]. In this case `errno` is set to `EDOM`.

Description:

`acos` calculates the arc cosine of a number.

See also:

`acosf`

`acosf` Calculates the arc cosine of a `float` number.

Synopsis:

```
#include <mathf.h>
float acosf(float x);
```

Arguments:

`float x`	A number in the range [−1..+1].

Results:

Returns the arc cosine of `x` in the range [0..pi] radians.

Errors:

A domain error occurs if `x` is not in the range [−1..+1]. In this case `errno` is set to `EDOM`.

Description:

`float` form of `acos`.

See also:

`acos`

`alloc86` Allocates a block of host memory. DOS only.

Synopsis:

```
#include <dos.h>
pcpointer alloc86(int n);
```

Arguments:

`int n` The number of bytes of host memory to be allocated.

Results:

Returns a pointer to the allocated block of host memory.

Errors:

Returns zero (0) if the allocation fails and sets `errno` to the value `EDOS`. Any attempt to use `from86` on systems other than DOS also sets `errno` to `EDOS`. Failure of the function also generates the server error message:

```
[Encountered unknown primary tag (50)]
```

Description:

`alloc86` allocates a block of memory on the DOS host and returns a pointer to it. If the memory cannot be allocated, a NULL pointer is returned. The allocated memory cannot be accessed directly by the transputer program but only by means of the functions `to86` and `from86`.

Note: Intel 80*x*86 architecture limits the amount of memory which can be contained in a single segment to 65536 bytes; `alloc86` cannot allocate more than this architectural limit.

See also:

`from86` `to86`

asctime Returns time from the tm structure as an ASCII string.

Synopsis:

```
#include <time.h>
char* asctime(const struct tm *timeptr);
```

Arguments:

`const struct tm *timeptr`	A pointer to the time structure to be converted.

Results:

Returns a pointer to the ASCII time string.

Errors:

None.

Description:

`asctime` returns the values in the `timeptr` structure as an ASCII string in the form:

Thu Nov 05 18:19:01 1987

The string pointed to may be overwritten by subsequent calls to `asctime`.

Example:

```
/* Displays the current time */

#include  <time.h>
#include  <stdio.h>

int main()
{
   struct tm *now;
   time_t clck;

   time(&clck); /* Get current time in secs */

   now = localtime( &clck);
                /* Convert time to
                   a structure (tm) */
```

```
        printf("The time is: %s\n", asctime(now));
    }
```

Note: Care should be taken when calling `asctime` in a concurrent environment. Calls to the function by independently executing, unsynchronised processes may corrupt the returned time value.

See also:

`ctime localtime strftime clock difftime mktime time`

`asin` Calculates the arc sine of the argument.

Synopsis:

```
#include <math.h>
double asin(double x);
```

Arguments:

`double x` A number in the range [-1..+1].

Results:

Returns the arc sine of `x` in the range [-pi/2..+pi/2] radians.

Errors:

A domain error occurs if `x` is not in the range [-1..+1]. In this case `errno` is set to `EDOM`.

Description:

`asin` calculates the arc sine of a number.

See also:

`asinf`

`asinf` Calculates the arc sine of a `float` number.

```
#include <mathf.h>
float asinf(float x);
```

Arguments:

`float x` A number in the range [−1..+1].

Results:

Returns the arc sine of `x` in the range [−pi/2..+pi/2] radians.

Errors:

A domain error occurs if `x` is not in the range [−1..+1]. In this case `errno` is set to `EDOM`.

Description:

`float` form of `asin`.

See also:

`asin`

`assert` Inserts diagnostic messages.

Synopsis:

```
#include <assert.h>
void assert(int expression);
```

Arguments:

`int expression`	The condition to be asserted.

Results:

Returns no value.

Errors:

None.

Description:

`assert` is a debugging macro. If it is called with `expression` equal to zero, `assert` terminates the program by calling `abort`. The action of `abort` when called by `assert` depends on the most recent call to `set_abort_action`.

If `expression` is non-zero, no action is taken.

If the function is linked with the full runtime library the following message is written to `stderr`:

`*** assertion failed:` *condition*, `file` *filename*, `line` *linenumber*

If the function is linked with the reduced runtime library then no message is displayed.

The definition of the `assert` macro depends upon the definition of the `NDEBUG` macro. If `NDEBUG` is defined before the definition of `assert` then `assert` is defined as:

```
#define assert(ignore) ((void)0)
```

If `assert` is defined first the definition is honoured and NDEBUG is ignored.

Example:

```
#include  <stdio.h>
#include  <assert.h>

float divide (float a, float b)
{
   assert(b == 0.0);
   return a/b;
}

int main( void )
{
   float res;

   res = divide(1.0F,2.0F);
   printf("1.0 divided by 2.0 is: %f\n",res);
   res = divide(1.0F,0.0F);
   printf("1.0 divided by 0.0 is: %f\n",res);
}
/*
 *   Output:
 *
 * *** assertion failed: b == 0.0,
 *     file assert.c, line 6
 *
 */
```

See also:

abort debug_assert

`atan` Calculates arc tangent.

Synopsis:

```
#include <math.h>
double atan(double x);
```

Arguments:

`double x` A number.

Results:

Returns the arctan of **x** in the range [-pi/2..+pi/2] radians.

Errors:

None.

Description:

`atan` calculates the arc tangent of a number.

See also:

`atanf`

atan2 Calculates the arc tangent of y/x.

Synopsis:

```
#include <math.h>
double atan2(double y, double x);
```

Arguments:

`double y`	The y value.
`double x`	The x value.

Results:

Returns the arc tangent of `y/x` in the range [-pi..+pi] radians.

Errors:

A domain error occurs if `x` and `y` are zero. In this case `errno` is set to `EDOM`.

Description:

`atan2` calculates the arc tangent of `y/x`.

See also:

`atan2f`

`atan2f` Calculates arc tangent of y/x where both are `floats`.

Synopsis:

```
#include <mathf.h>
float atan2f(float y, float x);
```

Arguments:

`float y`	The numerator.
`float x`	The denominator.

Results:

Returns the arc tangent of `y/x` in the range [–pi..+pi] radians.

Errors:

A domain error occurs if `x` and `y` are zero. In this case `errno` is set to `EDOM`.

Description:

`float` form of `atan2`.

`See also`: `atan2`

`atanf` Calculates the arc tangent of a `float` number.

Synopsis:

```
#include <mathf.h>
float atanf(float x);
```

Arguments:

`float x` A number.

Results: Returns the arc tangent of `x` in the range [–pi/2..+pi/2] radians.

Errors:

None.

Description: `float` form of `atan`.

See also:

`atan`

atexit Specifies a function to be called when the program ends.

Synopsis:

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

Arguments:

`void (*func)(void)` A pointer to the function to be called.

Results:

Returns zero if **atexit** is successful and non-zero if it is not.

Errors:

None.

Description:

atexit records that the function pointed to by **func** is to be called (without arguments) at normal termination of the program.

A maximum of 32 functions can be recorded for execution on exit. They will be called in reverse order of their being recorded (that is, last in, first out).

Note: In the parallel environment **atexit** works on program termination rather than process termination. A maximum of 32 functions can be registered as exit functions per program.

Example:

```
#include  <stdlib.h>
#include  <stdio.h>

void first_exit( void )
{
   printf("First_exit called on exit\n");
}

void second_exit( void )
{
   printf("Second_exit called on exit\n");
}
```

```
int main( void )
{
   atexit(second_exit);
   atexit(first_exit);
   printf("About to exit from program\n");
   return 0;
}

/*
 *   Output:
 *
 *          About to exit from program
 *          First_exit called on exit
 *          Second_exit called on exit
 *
 */
```

See also:

`exit`

`atof` Converts a string of characters to a `double`.

Synopsis:

```
#include <stdlib.h>
double atof(const char *nptr);
```

Arguments:

`const char *nptr`	A pointer to the string to be converted.

Results:

Returns the converted value.

Errors:

If the string cannot be converted, `atof` returns 0 (zero). If the conversion would cause overflow or underflow in the `double` value, the behaviour is undefined.

Description:

`atof` converts the string pointed to by `nptr` to a double precision floating point number. `atof` expects the string to consist of:

1. Leading white space (optional).
2. A plus or minus sign (optional).
3. A sequence of decimal digits, which may contain a decimal point.
4. An exponent (optional) consisting of an 'E' or 'e' followed by an optional sign and a string of decimal digits.
5. One or more unrecognised characters (including the NULL string terminating character).

`atof` ignores the leading white space, and converts all the recognised characters. If there is no decimal point or exponent part in the string, a decimal point is assumed after the last digit in the string.

The string is invalid if the first non-space character in the string is not one of the following characters: + – 0 1 2 3 4 5 6 7 8 9

Example:

```
#include  <stdio.h>
#include  <stdlib.h>

int main()
{
 char *array;
 double x;


 array = "  -4235.120E-3";
 x = atof(array);
 printf("Float = %f\n", x);


 array = "  -735492.45";
 x = atof(array);
 printf("Float = %e\n", x);
}
/*
Prints Float = -4.235120
       Float = -7.354924e+05
*/
```

See also:

atoi atol strtod

`atoi` Converts a string to an int.

Synopsis:

```
#include <stdlib.h>
int atoi(const char *nptr);
```

Arguments:

`const char *nptr` A pointer to the string to be converted.

Results:

Returns the converted value.

Errors:

If the string cannot be converted, `atoi` returns 0. If the conversion would overflow or underflow, the behaviour is undefined.

Description:

`atoi` converts the string pointed to by `nptr` to an integer. `atoi` expects the string to consist of:

1. Leading white space (optional).
2. A plus or minus sign (optional).
3. A sequence of decimal digits.
4. One or more unrecognised characters (including the NULL string terminating character).

`atoi` ignores the leading white space, and converts all the recognised characters.

The string is invalid if the first non-space character in the string is not one of the following characters: + – 0 1 2 3 4 5 6 7 8 9

Example:

```
#include  <stdlib.h>
#include  <stdio.h>

int main( void )
{
   char *array;
   int x;
```

```
        array = "  -4235";
        x = atoi(array);
        printf("Integer is: %d\n", x);

        array = "-735492 and some rubbish text";
        x = atoi(array);
        printf("Integer is: %d\n", x);
    }

    /*
     *   Output:
     *
     *           Integer is: -4235
     *           Integer is: -735492
     *
     */
```

See also:

`atof atol strtol`

`atol` Converts a string to a long integer.

Synopsis:

```
#include <stdlib.h>
long int atol(const char *nptr);
```

Arguments:

`const char *nptr` A pointer to the string to be converted.

Results:

Returns the converted value.

Errors:

If the string cannot be converted, `atol` returns 0. If the conversion would overflow or underflow, the behaviour is undefined.

Description:

`atol` converts the string pointed to by `nptr` to a long integer. `atol` expects the string to consist of:

1. Leading white space (optional).
2. A plus or minus sign (optional).
3. A sequence of decimal digits.
4. One or more unrecognised characters (including the NULL string terminating character).

`atol` ignores the leading white space, and converts all the recognised characters.

The string is invalid if the first non-space character in the string is not one of the following characters: + – 0 1 2 3 4 5 6 7 8 9

Example:

```
#include  <stdio.h>
#include  <stdlib.h>

int main()
{
 char *array;
 long l;
```

```
     array = "  -735492.45";
     l = atol(array);
     printf("Long = %ld\n", l);
    }

    /*
    Prints "Long = -735492"
    */
```

See also:

`atof atoi strtod strtol`

`bdos` Performs a simple DOS function. DOS only.

Synopsis:

```
#include <dos.h>
int bdos(int dosfn, int dosdx, int dosal);
```

Arguments:

`int dosfn`	Value to assign to the **ah** register.
`int dosdx`	Value to assign to the **dx** register.
`int dosal`	Value to assign to the **al** register.

Results:

Returns the value of the **ax** register.

Errors:

Returns zero (0) on error and sets `errno` to the value `EDOS`. Any attempt to use `bdos` on operating systems other than DOS also sets `errno` to `EDOS`. Failure of the function also generates the server error message:

```
[Encountered unknown primary tag (50)]
```

Description:

`bdos` performs a DOS function call interrupt on the host with the `ah` register (specifying the DOS function call number) set to `dosfn`, and with the `dx` and `al` registers set to `dosdx` and `dosal` respectively. It is a shorthand form of `int86` for the very simplest DOS function calls only.

`bdos` is not included in the reduced library.

See also:

`intdos` `int86`

bsearch Searches a sorted array for a given object.

Synopsis:

```
#include <stdlib.h>
void *bsearch(const void *key,
              const void *base,
              size_t nmemb, size_t size,
              int (*compar)(const void *,
                            const void *));
```

Arguments:

`const void *key`	A pointer to the object to be matched.
`const void *base`	A pointer to the start of the array.
`size_t nmemb`	The number of objects in the array.
`size_t size`	The size of the array objects.
`int (*compar) (const void *, const void *)`	A pointer to the comparison function.

Results:

Returns a pointer to the object if found; otherwise **bsearch** returns a null pointer. If more than one object in the array matches the key, it is not defined which one the return value points to.

Errors:

None.

Description:

bsearch searches the array pointed to by **base** for an object which matches the object pointed to by **key**. The array contains **nmemb** objects of **size** bytes.

The objects are compared using the comparison function pointed to by **compar**. The function must return an integer less than, equal to, or greater than zero, depending on whether the first argument to the function is considered to be less than, equal to, or greater than the second argument.

The base array must already be sorted in ascending order (according to the comparison performed by the function pointed to by **compar**).

Example:

```
/*
 * Receives a list of arguments from the
 * terminal, and searches them for the
 * string "findme".
 */

#include  <stdio.h>
#include  <stdlib.h>
#include  <string.h>

int compare(const void *arg1, const void *arg2)
{
   return(strncmp(*(char **)arg1, *(char **)arg2,
          strlen(*(char **)arg1)));
}


int main(int argc, char *argv[])
{
   char **result;
   char *key = "findme";

   /* sort the command line arguments according
      to the string compare function 'compare' */

   qsort(argv, argc, sizeof(char *), compare);

   /* Find the argument which starts with
      the string in 'key'  */

   result = (char **)bsearch(&key, argv, (size_t)argc,
                          sizeof(char *), compare);

   if (result != NULL)
     printf("\n'%s' found\n", *result);
   else
     printf("\n'%s' not found\n", key);
}
```

See also:

qsort

calloc

Allocates memory space for an array of items and initialises the space to zeros.

Synopsis:

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

Arguments:

`size_t nmemb`	The number of items in the array to be allocated.
`size_t size`	The size of the array items.

Results:

Returns a pointer to the allocated space if the allocation is successful; otherwise `calloc` returns a null pointer. If either argument is zero `calloc` returns a NULL pointer.

Errors:

`calloc` returns a null pointer if there is not enough free space in memory.

Description:

`calloc` allocates space in memory for an array containing `nmemb` items, where each item is `size` bytes long. The allocated memory is initialised to zeros.

Programming note: On the T2 family of transputers pointers should always be initialised explicitly, because the NULL pointer on these machines is represented by a non-zero bit pattern.

See also:

`free malloc realloc`

`ceil` Calculates the smallest integer not less than the argument.

Synopsis:

```
#include <math.h>
double ceil(double x);
```

Arguments:

`double x` A number.

Results:

Returns the smallest integer (expressed as a double) which is not less than `x`.

Errors:

None.

Description:

`ceil` calculates the smallest integer which is not less than `x`.

See also:

`floor` `ceilf`

`ceilf` `float` form of `ceil`.

Synopsis:

```
#include <mathf.h>
float ceilf(float x);
```

Arguments:

`float x` A number.

Results:

Returns the smallest integer (expressed as type `float`) which is not less than `x`.

Errors:

None.

Description:

`float` form of `ceil`.

See also:

`ceil`

ChanAlloc Allocates and initialises a channel.

Synopsis:

```
#include <channel.h>
Channel *ChanAlloc(void);
```

Arguments:

None.

Results:

Returns a pointer to an initialised channel, or NULL if the space could not be allocated.

Errors:

Returns NULL if space could not be allocated.

Description:

Allocates and initialises a channel.

Note: All channels *must* be allocated (by a call to `ChanAlloc` or or by specific allocation of memory space) before use.

See also:

`ChanReset`

ChanIn Inputs data on a channel.

Synopsis:

```
#include <channel.h>
void ChanIn(Channel *c, void *cp, int count);
```

Arguments:

`Channel *c`	A pointer to the input channel.
`void *cp`	A pointer to the array where the data will be stored.
`int count`	The number of bytes of data.

Results:

Returns no result.

Errors:

None.

Description:

Inputs `count` bytes of data on the specified channel and stores them in the array pointed to by `cp`.

See also:

`ChanOut ChanInInt ChanInChar ChanInChanfail`
`ChanInTimeFail`

ChanInChanFail Inputs data on a link channel or aborts.

Synopsis:

```
#include <channel.h>
int ChanInChanFail(Channel *chan, void *cp,
                   int count, Channel *failchan);
```

Arguments:

`Channel *c`	A pointer to the input channel.
`void *cp`	A pointer to an array where the data will be stored.
`int count`	The number of bytes of data.
`Channel *failchan`	A pointer to the channel on which the failure message is received.

Results:

Returns zero (0) if communication completes, one (1) if communication is aborted by a message on the failure channel.

Errors:

None.

Description:

`ChanInChanFail` is used to perform reliable channel communication on a link. The function inputs `count` bytes of data on the specified channel into the array pointed to by `cp`. It can be aborted by an integer, and only an integer, passed on `failchan`. Typically `failchan` will be a channel from a process which is monitoring the integrity of the link.

See also:

`ChanIn` `ChanInTimeFail`

ChanInChar Inputs one byte on a channel.

Synopsis:

```
#include <channel.h>
char ChanInChar(Channel *c);
```

Arguments:

`Channel *c`	A pointer to the input channel.

Results:

Returns the input byte.

Errors:

None.

Description:

Inputs a single byte on a channel.

See also:

`ChanOutChar` `ChanIn`

`ChanInInt` Inputs an integer on a channel.

Synopsis:

```
#include <channel.h>
int ChanInInt(Channel *c);
```

Arguments:

`Channel *c`	A pointer to the input channel.

Results:

Returns the input integer.

Errors:

None.

Description:

Inputs a single integer on a channel.

See also:

`ChanOutInt` `ChanIn`

`ChanInit` Initialises a channel pointer.

Synopsis:

```
#include <channel.h>
void ChanInit(Channel *chan);
```

Arguments:

`Channel *chan`	A pointer to a channel.

Results:

Returns no result.

Errors:

None.

Description:

Initialises the channel pointed to by `chan` to the value `NotProcess_p`. `NotProcess_p` is defined in `channel.h`.

Example:

```
#include <channel.h>
#include <stdlib.h>

Channel c1, *c2;

ChanInit(&c1);
c2 = (Channel *)malloc(sizeof(Channel));
ChanInit(c2);
```

See also:

`ChanReset`

ChanInTimeFail Inputs data on a channel or times out.

Synopsis:

```
#include <channel.h>
int ChanInTimeFail(Channel *chan, void *cp,
                   int count, int time);
```

Arguments:

`Channel *c`	A pointer to the input channel.
`void *cp`	A pointer to an array where the data will be stored.
`int count`	The number of bytes of data.
`int time`	The time after which the communication is aborted if no input occurs.

Results:

Returns zero (0) if the communication is successful, one (1) if timeout occurs before the communication completes.

Errors:

None.

Description:

`ChanInTimeFail` is used to timeout channel communication on a link. It inputs `count` bytes of data on the specified channel and stores them in the array pointed to by `cp`, or aborts if the transputer clock reaches the specified time. Typically it is used to notify delay on a link so that the communication can be routed elsewhere.

See also:

`ChanIn` `ChanInChanFail` `ChanOutTimeFail`

ChanOut Outputs data on a channel.

Synopsis:

```
#include <channel.h>
void ChanOut(Channel *c, void *cp, int count);
```

Arguments:

`Channel *c`	A pointer to the output channel.
`void *cp`	A pointer to an array containing the output data.
`int count`	The number of bytes of data.

Results:

Returns no result.

Errors:

None.

Description:

Outputs `count` bytes of data on the channel `c`. The data is taken from the array pointed to by `cp`.

See also:

`ChanIn ChanOutInt ChanOutChar`

ChanOutChanFail Outputs data or aborts on failure.

Synopsis:

```
#include <channel.h>
int ChanOutChanFail(Channel *chan, void *cp,
                    int count, Channel *failchan);
```

Arguments:

`Channel *c`	A pointer to the output channel.
`void *cp`	A pointer to an array containing the output data.
`int count`	The number of bytes of data.
`Channel *failchan`	A pointer to the channel on which the failure message is received.

Results:

Returns zero (0) if communication completes normally, one (1) if communication is aborted by a message on the failure channel.

Errors:

One.

Description:

`ChanOutChanFail` is used to perform reliable channel communication on a link. It outputs `count` bytes of data on the specified channel from the array pointed to by `cp`. The function can be aborted by an integer, and only an integer, passed on the channel `failchan`. Typically `failchan` will be a channel from a process which is monitoring the integrity of the link.

See also:

`ChanOut` `ChanOutTimeFail`

`ChanOutChar` Outputs one byte on a channel.

Synopsis:

```
#include <channel.h>
void ChanOutChar(Channel *c, char ch);
```

Arguments:

`Channel *c`	A pointer to the output channel.
`char ch`	The byte to be output.

Results:

Returns no result.

Errors:

None.

Description:

Outputs a single byte on a channel.

See also:

`ChanInChar` `ChanOut`

`ChanOutInt` Outputs an integer on a channel.

Synopsis:

```
#include <channel.h>
void ChanOutInt(Channel *c, int n);
```

Arguments:

`Channel *c`	A pointer to the output channel.
`int n`	The integer to be output.

Results:

Returns no result.

Errors:

None.

Description:

Outputs a single integer on a channel.

See also:

`ChanOutInt ChanIn`

`ChanOutTimeFail` Outputs data on a channel or times out.

Synopsis:

```
#include <channel.h>
int ChanOutTimeFail(Channel *chan, void *cp,
                    int count, int time);
```

Arguments:

`Channel *c`	A pointer to the output channel.
`void *cp`	A pointer to an array containing the output data.
`int count`	The number of bytes of data.
`int time`	The time after which the communication is aborted if no output occurs.

Results:

Returns zero if the communication is successful, one (1) if timeout occurs before the communication completes.

Errors:

None.

Description:

`ChanOutTimeFail` is used to timeout channel communication on a link. It outputs `count` bytes of data on the specified channel from the array pointed to by `cp`. The functions aborts if the transputer clock reaches the specified time before the communication takes place. Typically it is used to notify delay on a link so that the communication can be routed elsewhere.

See also:

`ChanOut` `ChanOutChanFail`

ChanReset Resets a channel.

Synopsis:

```
#include <channel.h>
int ChanReset(Channel *c);
```

Arguments:

`Channel *c`	A pointer to the channel to be reset.

Results:

Returns either `NotProcess_p`, or a process descriptor.

Errors:

None.

Description:

Resets a channel to the value `NotProcess_p` and returns the process descriptor of the channel waiting to communicate, or `NotProcess_p`. If the value returned is `NotProcess_p`, no process was waiting on the channel, and any communication on that channel had completed successfully.

`NotProcess_p` is defined in `channel.h`.

See also:

`ChanInit`

`clearerr` Clears error and end-of-file indicators for a file stream.

Synopsis:

```
#include <stdio.h>
void clearerr(FILE *stream);
```

Arguments:

`FILE *stream`	A pointer to a file stream.

Results:

Returns no value.

Errors:

None.

Description:

`clearerr` clears the error and end-of-file indicators for a file stream.

See also:

`rewind`

`clock` Determines the amount of processor time used.

Synopsis:

```
#include <time.h>
clock_t clock(void);
```

Arguments:

None.

Results:

Returns the time used by the program since it started. If the processor time is not available or the value cannot be represented, the value (`clock_t`)-1 is returned.

Errors:

If the processor time is not available or the value cannot be represented, the value (`clock_t`)-1 is returned.

Description:

`clock` returns the processor time used by the program since it started. The exact interval returned extends from the time the `main` function was called until program termination.

To obtain the time in seconds the return value should be divided by `CLOCKS_PER_SEC`.

See also:

`asctime ctime localtime strftime difftime mktime time`

`close` Closes a file. File handling primitive.

Synopsis:

```
#include <iocntrl.h>
int close(int fd);
```

Arguments: `int fd` File descriptor of the file to be closed.

Results:

Returns 0 if successful or −1 on error.

Errors:

If an error occurs `close` sets `errno` to the value `EIO`.

Description:

`close` is the low level file close function used by `fclose`. It takes a file descriptor as a parameter instead of a FILE pointer. The file descriptor will usually have been returned by the `open` or `creat` functions.

`close` is not included in the reduced library.

`cos` Calculates the cosine of the argument.

Synopsis:

```
#include <math.h>
double cos(double x);
```

Arguments:

`double x` A number in radians.

Results:

Returns the cosine of `x` in radians.

Errors:

None.

Description:

`cos` calculates the cosine of a number.

See also:

`cosf`

`cosf` Calculates the cosine of a `float` number.

Synopsis:

```
#include <mathf.h>
float cosf(float x);
```

Arguments:

`float x` A number in radians.

Results:

Returns the cosine of `x` in radians.

Errors:

None.

Description:

`float` form of `cos`.

See also:

`cos`

`cosh` Calculates the hyperbolic cosine of the argument.

Synopsis:

```
#include <math.h>
double cosh(double x);
```

Arguments:

`double x` A number.

Results:

Returns the hyperbolic cosine of `x`.

Errors:

A range error will occur if `x` is so large that `cosh` would result in an overflow. In this case `cosh` returns the value `HUGE_VAL` (with the same sign as the correct value of the function) and `errno` is set to `ERANGE`.

Description:

`cosh` calculates the hyperbolic cosine of a number.

See also:

`coshf`

`coshf` Calculates the hyperbolic cosine of a `float` number.

Synopsis:

```
#include <mathf.h>
float coshf(float x);
```

Arguments:

`float x` A number.

Results:

Returns the hyperbolic cosine of `x`.

Errors:

A range error will occur if `x` is so large that `coshf` would result in an overflow. In this case `coshf` returns the value `HUGE_VAL_F` (with the same sign as the correct value of the function) and `errno` is set to `ERANGE`.

Description:

`float` form of `cosh`.

See also:

`cosh`

`creat` Creates a file for writing. File handling primitive.

Synopsis:

```
#include <iocntrl.h>
int creat(char *name, int flag);
```

Arguments:

`char *name`	The name of the file to be created.
`int flag`	A number which specifies the mode in which the file is opened.

Results:

Returns a file descriptor for the file, or −1 on error.

Errors:

If an error occurs `creat` sets `errno` to the value `EIO`.

Description:

`creat` creates a file with filename `name` and opens it in 'write' and 'truncate' modes. If the file already exists, and if the host system permits, the file is overwritten.

The value of `flag` determines how the file is opened. It can take two values, as follows:

`O_BINARY`	Open file in binary mode.
`O_TEXT`	Open file as a text file.

The default is to open the file as a text file.

`creat` has the same effect as a call to `open` with the following parameters:

```
open(name, O_WRONLY | O_TRUNC | flag);
```

`creat` is not included in the reduced library.

See also:

`open`

`ctime` Converts a `time_t` value to a string.

Synopsis:

```
#include <time.h>
char *ctime(const time_t *timer);
```

Arguments:

`const time_t *timer` A pointer to a location containing a time.

Results:

Returns a pointer to a string describing the local time.

Errors:

None.

Description:

`asctime` converts the value pointed to by `timer` to a `tm` structure, and then writes the contents of the structure into a string in the following form:

Thu Nov 05 18:19:01 1987

Example:

```
/* Displays the current time */
#include  <time.h>
#include  <stdio.h>

int main( void )
{
   time_t now;

   time(&now);
   printf("The time is: %s\n",ctime(&now));
}
```

`ctime` is equivalent to the following call to `asctime`:

```
asctime (localtime(timer));
```

See also: `asctime localtime strftime clock difftime mktime time gmtime`

debug_assert Stops process/alerts debugger if condition fails.

Synopsis:

```
#include <misc.h>
void debug_assert(const int exp);
```

Arguments:

`const int exp`	An integer expression for the condition to be asserted.

Results:

Returns no result.

Errors:

None.

Description:

`debug_assert` replaces `assert` for programs that will be debugged in breakpoint mode. If `expression` evaluates FALSE `debug_assert` stops the process and sends process data to the debugger. If `expression` evaluates TRUE no action is taken.

If the program is not being run within the breakpoint debugger and the assertion fails, then the function behaves like `debug_stop`.

See also:

`assert debug_message debug_stop`

`debug_message` Inserts a debugging message.

Synopsis:

```
#include <misc.h>
void debug_message(const char *message);
```

Arguments:

`const char *message`	The text of the message.

Results:

Returns no result.

Errors:

None.

Description:

`debug_message` sends a message to the debugger which is displayed along with normal program output.

If the program is not being run within the breakpoint debugger the function has no effect.

See also:

`debug_assert` `debug_stop`

`debug_stop` Stops a process and notifies the debugger.

Synopsis:

```
#include <misc.h>
void debug_stop(void);
```

Arguments:

None.

Results:

Returns no result.

Errors:

None.

Description:

`debug_stop` stops the process and sends process data to the debugger. If the program is in HALT mode the processor halts and any other processes running on that processor are also stopped.

If the program is not being run within the breakpoint debugger then the function stops the process or processor, depending on the error mode in which the processor is executing.

See also:

`debug_assert` `debug_message`

difftime Calculates the difference between two times.

Synopsis:

```
#include <time.h>
double difftime(time_t time1, time_t time0);
```

Arguments:

`time_t time1`	The first time.
`time_t time0`	The second time.

Results:

Returns the difference, in seconds, between `time1` and `time0`.

Errors:

None.

Description:

`difftime` calculates the difference in time between `time1` and `time0` (`time1` - `time0`).

See also:

`asctime ctime localtime strftime clock mktime time gmtime`

`div` Calculates the quotient and remainder of a division.

Synopsis:

```
#include <stdlib.h>
div_t div(int numer, int denom);
```

Arguments:

`int numer`	The numerator.
`int denom`	The denominator.

Results:

Returns a structure of type `div_t` which consists of the quotient and remainder. The structure contains:

`int quot`	The quotient.
`int rem`	The remainder.

Errors:

If the result cannot be represented the behaviour of `div` is undefined.

Description:

`div` calculates the quotient and remainder formed by dividing the numerator `num` by the denominator `denom`.

See also:

`ldiv`

`exit` Terminates a program.

Synopsis:

```
#include <stdlib.h>
void exit(int status);
```

Arguments:

`int status` A value to be passed back to the calling environment.

Results:

`exit` does not return.

Errors:

None.

Description:

`exit` causes normal program termination. The actions taken are as follows:

1. The functions recorded by `atexit` are called in reverse order.
2. All open output streams are flushed.
3. All open streams are closed.
4. All files created by `tmpfile` are removed.
5. Control is returned to the host environment.

The value of `status` signals success or failure of the termination operation to the the host environment. If `status` is zero or equal to `EXIT_SUCCESS` the termination was successful; if `status` is equal to `EXIT_FAILURE` the termination was unsuccessful. If `status` is other any value than zero, `EXIT_SUCCESS` or `EXIT_FAILURE`, the status returned is the numerical value of the argument. `EXIT_SUCCESS` and `EXIT_FAILURE` are declared in the header file `stdlib.h`.

When used in a configured process `exit` does not terminate the server. To terminate the server from a configured process use `exit_terminate`.

Caution: `exit` should not be called from a C function that is running *in parallel* with any other function. The effect on the program may be unpredictable.

Example:

```
#include  <stdlib.h>
#include  <stdio.h>

int main( void )
{
   printf("About to do an exit\n");
   exit(EXIT_SUCCESS);
   printf("Not printed\n");
}
```

See also:

atexit exit_repeat exit_terminate

`exit_repeat` Terminates a program so that it can be restarted.

Synopsis:

```
#include <misc.h>
void exit_repeat(int status);
```

Arguments:

`int status` A value to be passed back to the calling environment.

Results:

Returns no result.

Errors:

None.

Description:

`exit_repeat` terminates the C program and returns its argument to the calling environment. Unlike `exit`, `exit_repeat` retains the program and allows it to be rerun without rebooting the transputer.

Only programs which consist of a single C program running on a single transputer, and which have been made bootable using the collector 'T' option, can be repeat invoked. In all other cases `exit_repeat` acts like `exit`.

Caution: `exit_repeat` should not be called from a C function that is running *in parallel* with any other function. The effect on the program may be unpredictable.

The first element of the `argv` array is lost in the process of calling `exit_repeat`. Therefore programs that read the program name from the first element of the array will need to be rebooted.

Note: If use is made of the predefined constants `EXIT_FAILURE` or `EXIT_SUCCESS` then the header file `stdlib.h` must be included.

See also:

`exit`

exit_terminate Version of exit for configured processes.

Synopsis:

```
#include <misc.h>
void exit_terminate(int status);
```

Arguments:

`int status`	A value to be passed back to the calling environment.

Results:

Returns no result.

Errors:

None.

Description:

`exit_terminate` is the equivalent of `exit` for a configured process (one which has been placed on a processor by `icconf`).

`exit_terminate` works in the same way as `exit` by passing a single argument back to the calling environment. The argument only reaches the calling environment if the server is terminated.

`exit_terminate` only works for *configured* programs linked with the *full* run-time library. In all other cases it acts like `exit`.

Note: If use is made of the predefined constants `EXIT_FAILURE` or `EXIT_SUCCESS` then the header file `stdlib.h` must be included.

See also:

`exit` `exit_repeat`

exp Calculates the exponential function of the argument.

Synopsis:

```
#include <math.h>
double exp(double x);
```

Arguments:

double x A number.

Results:

Returns the exponential function of **x**.

Errors:

A range error occurs if the result of raising e to the power of **x** would cause overflow. In this case **exp** returns the value **HUGE_VAL** (with the same sign as the correct value of the function) and **errno** is set to **ERANGE**.

Description:

exp calculates the value of the constant e (2.71828...) raised to the power of a number.

See also:

expf

`expf` Calculates the exponential function of a `float` number.

Synopsis:

```
#include <mathf.h>
float expf(float x);
```

Arguments:

`float x` A number.

Results:

Returns the exponential function of `x`.

Errors:

A range error occurs if the result of raising e to the power of `x` would cause overflow. In this case `expf` returns the value `HUGE_VAL_F` (with the same sign as the correct value of the function) and `errno` is set to `ERANGE`.

Description:

`float` form of `exp`.

See also:

`exp`

`fabs` Calculates the absolute value of a floating point number.

Synopsis:

```
#include <math.h>
double fabs(double x);
```

Arguments:

`double x`	A number.

Results:

Returns the absolute value of the argument.

Errors:

None.

Description:

`fabs` calculates the absolute value of a number.

See also:

`fabsf`

`fabsf` Calculates the absolute value of a `float` number.

Synopsis:

```
#include <mathf.h>
float fabsf(float x);
```

Arguments:

`float x` A number.

Results:

Returns the absolute value of the argument.

Errors:

None.

Description:

`float` form of `fabs`.

See also:

`fabs`

`fclose` Closes a file stream.

Synopsis:

```
#include <stdio.h>
int fclose(FILE *stream);
```

Arguments:

`FILE *stream`	A pointer to the file stream.

Results:

Returns zero if the close was successful and `EOF` if it was not.

Errors:

None.

Description:

`fclose` closes the file stream pointed to by `stream`. The stream and any associated buffers are flushed. Any buffer which was allocated by the I/O system is deallocated.

Buffer data which is waiting to be written is sent to the host environment for writing to the file. Buffer data which is waiting to be read is ignored.

`fclose` is called automatically when `exit` is called.

`fclose` is not included in the reduced library.

See also:

`fopen`

`feof` Tests for End-Of-File.

Synopsis:

```
#include <stdio.h>
int feof(FILE *stream);
```

Arguments:

`FILE *stream`	A pointer to a file stream.

Results:

Returns zero if the End-Of-File indicator for `stream` is clear, non-zero if it is set.

Errors:

None.

Description:

`feof` tests the state of the End-Of-File indicator for the file stream `stream`. It returns zero if the indicator is clear, and non-zero if it is set.

`feof` is not included in the reduced library.

See also:

`ferror`

`ferror` Tests for a file error.

Synopsis:

```
#include <stdio.h>
int ferror(FILE *stream);
```

Arguments:

`FILE *stream` A pointer to a file stream.

Results:

Returns zero if the error indicator for `stream` is clear, and non-zero if it is set.

Errors:

None.

Description:

`ferror` tests the state of the error indicator for the file stream `stream`. It returns zero if the error indicator is clear, and non-zero if it is set.

`ferror` is not included in the reduced library.

See also:

`feof`

`fflush` Flushes an output stream.

Synopsis:

```
#include <stdio.h>
int fflush(FILE *stream);
```

Arguments:

`FILE *stream`	A pointer to the stream to be flushed.

Results:

Returns `EOF` if a write error occurred, otherwise 0.

Errors:

If a write error occurs, `fflush` returns EOF.

Description:

If `stream` points to an output stream, `fflush` causes any outstanding data for the stream to be written to the file. The behaviour is undefined for a stream which is neither open for output nor update.

If `stream` is NULL `fflush` flushes all streams that are open for output.

`fflush` is not included in the reduced library.

See also:

`ungetc`

`fgetc` Reads a character from a file stream.

Synopsis:

```
#include <stdio.h>
int fgetc(FILE *stream);
```

Arguments:

`FILE *stream`	A pointer to a file stream.

Results:

Returns the next character from the file stream.

Errors:

If the stream is at End-Of-File, the end-of-file indicator for the stream is set and `fgetc` returns EOF. If a read error occurs, the error indicator for the stream is set and `fgetc` returns EOF.

Description:

`fgetc` returns the next character from the opened file identified by the file stream pointer `stream`, and advances the read/write position indicator for the file stream.

`fgetc` is not included in the reduced library.

See also:

`fgets fputc getc ungetc`

`fgetpos` Gets the position of the read/write file pointer.

Synopsis:

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

Arguments:

`FILE *stream`	A pointer to a file stream.
`fpos_t *pos`	A pointer to an object where the current value of the read/write file pointer can be stored.

Results:

Returns zero if the operation was successful. If the operation fails `fgetpos` sets `errno` to `EFILPOS` and returns non-zero.

Errors:

If the operation was unsuccessful, `fgetpos` returns a non-zero value.

Description:

`fgetpos` stores the position of the read/write pointer of the file stream `stream` in the object pointed to by `pos`. This information is in a form usable by the `fsetpos` function.

`fgetpos` is not included in the reduced library.

See also:

`fsetpos`

`fgets` Reads a line from a file stream.

Synopsis:

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

Arguments:

`char *s`	A pointer to a buffer to receive the string.
`int n`	The size of the array.
`FILE *stream`	A pointer to a file stream.

Results:

Returns `s` if successful. If end-of-file is encountered before a character is read, or a read error occurs, `fgets` returns a NULL pointer.

Errors:

`fgets` returns a NULL pointer if end-of-file is encountered before a character is read, or a read error occurs.

Description:

`fgets` reads a string of a maximum (n−1) characters from the file stream identified by `stream`. `fgets` stops reading when it encounters a newline character or an end-of-file character. A string terminating character is written into the array after the last character read. The newline character forms part of the string.

`fgets` is not included in the reduced library.

See also:

`fgetc fputs gets`

`filesize` Determines the size of a file. File handling primitive.

Synopsis:

```
#include <iocntrl.h>
long int filesize(int fd);
```

Arguments:

`int fd` A file descriptor.

Results:

Returns the size of the file in bytes or -1 on error.

Errors:

If an error occurs `filesize` sets `errno` to the value `EIO`.

Description:

`filesize` takes a file descriptor and returns the size of the file in bytes. If the file is open for writing, `filesize` returns the current size of the file.

`filesize` is not included in the reduced library.

floor Calculates the largest integer not greater than the argument.

Synopsis:

```
#include <math.h>
double floor(double x);
```

Arguments:

`double x` A number.

Results:

Returns the largest integer (expressed as a double) which is not greater than `x`.

Errors:

None.

Description:

`floor` calculates the largest integer which is not greater than `x`.

See also:

`ceil` `floorf`

`floorf` `float` form of `floor`.

Synopsis:

```
#include <mathf.h>
int floorf(float x);
```

Arguments:

`float x`	A number.

Results:

Returns the largest integer (expressed as a `float`) which is not greater than `x`.

Errors:

None.

Description:

`float` form of `floor`.

See also:

`ceilf` `floor`

fmod Calculates the floating point remainder of x/y.

Synopsis:

```
#include <math.h>
double fmod(double x, double y);
```

Arguments:

`double x`	The dividend.
`double y`	The divisor.

Results:

Returns (with the same sign as `x`) the floating point remainder of `x/y`. If `y` is zero `errno` obtains the value `EDOM` and `fmod` returns zero.

Errors:

A domain error occurs if `y` is zero, and the function then returns zero. A range error occurs if the result is not representable.

Description:

`fmod` calculates the floating point remainder of `x/y`.

See also:

`fmodf`

`fmodf` Calculates the floating point remainder of x/y.

Synopsis:

```
#include <mathf.h>
float fmodf(float x, float y);
```

Arguments:

`float x`	The dividend.
`float y`	The divisor.

Results:

Returns (with the same sign as `x`) the floating point remainder of `x/y`. If `y` is zero `errno` obtains the value `EDOM` and `fmodf` returns zero.

Errors:

A domain error occurs if `y` is zero and a range error occurs if the result is not representable.

Description:

`float` form of `fmod`.

See also:

`fmod`

fopen Opens a file.

Synopsis:

```
#include <stdio.h>
FILE *fopen(const char *filename,
            const char *mode);
```

Arguments:

`char *filename`	The name of the file to be opened.
`const char *mode`	A string which specifies the mode in which the file is to be opened.

Results:

Returns a file pointer to the stream associated with the newly opened file. `fopen` returns a null pointer if it cannot open the file.

Errors:

If a file opened for reading does not exist or the open operation fails for any other reason, `fopen` returns a null pointer.

Description:

`fopen` opens the file named by the string pointed to by `filename`, in the mode specified by the `mode` string.

`fopen` is not included in the reduced library.

The following are valid mode strings:

"r"	Opens a text file for reading.
"w"	Opens a text file for writing. If the file already exists it is truncated to zero length. If the file does not exist, it is created.
"a"	Opens a text file for appending. If the file does not exist, it is created.
"rb"	Opens a binary file for reading.
"wb"	Opens a binary file for writing. If the file already exists it is truncated to zero length. If the file does not exist, it is created.
"ab"	Opens a binary file for appending. If the file does not exist, it is created.
"r+"	Opens a text file for reading and writing.
"w+"	Creates a text file for reading and writing. If the file exists, it is truncated to zero length.
"a+"	Opens a text file for reading, and writing at the end of the file. If the file does not exist, it will be created.
"r+b" or "rb+"	Opens a binary file for reading and writing.
"w+b" or "wb+"	Creates a binary file for reading and writing. If the file exists, it is truncated to zero length.
"a+b" or "ab+"	Opens a binary file for reading and writing at the end of the file. If the file does not exist, it will be created.

File output must not be followed by file input without an intervening call to `fflush` or one of the file positioning functions `fseek`, `fsetpos` and `rewind`. Similarly, input must not be followed by output without an intervening call to one of these functions unless EOF is encountered.

If a file is opened with a "+" in the mode string (opened for update), the file can be read from and written to without closing and reopening the file. However, you must call `fflush`, `fseek`, `fsetpos` or `rewind` between read and write operations.

Example:

```
#include <stdio.h>

int main( void )
{
   FILE *stream;

   stream = fopen("data.dat","r");

   if (stream == NULL)
      printf("Can't open data.dat file for
```

```
                  read\n");
        else
           printf("data.dat opened for read\n");
      }
```

See also:

fclose fflush freopen fseek fsetpos rewind

`fprintf` Writes a formatted string to a file.

Synopsis:

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format, ...);
```

Arguments:

`FILE *stream`	A pointer to an output file stream.
`const char *format`	An array of characters specifying the format.
`...`	Subsequent arguments to the format string.

Results:

Returns the number of characters written, or a negative value if an output error occurs.

Errors:

Returns a negative value if an output error occurs.

Description:

`fprintf` writes the string pointed to by `format` to the file stream `stream`. When `fprintf` encounters a percent sign % in the string, it expands the corresponding argument into the format defined by the format tokens after the sign.

`fprintf` is not included in the reduced library.

The format tokens consist of the following items:

1. Flags (optional):

–	causes the output to be left-justified in its field.
+	causes the output to start with a '+' or '–'.
' ' (blank space)	causes the output to start with a space if positive, and a '–' if negative. If the space and + flags appear together, the space flag is ignored.
#	causes: - an octal number to begin with 0. - a hex number to begin with 0x, or 0X for the x or X conversion specifiers. - a floating point number to contain a decimal point in (e, E, f, G, g,).
0	For d,i,o,u,x,X,e,E,f,g,G, conversions (see below), leading zeros are used to pad the fieldwidth. If both 0 and – flags both appear, the 0 is ignored. For d,i,o,u,x,X conversions, if a precision is specified the 0 flag is ignored.

2. Minimum width (optional):

The width is an integer constant which defines the minimum number of characters displayed. If the integer constant is replaced by an asterisk ('*'), an `int` argument supplies the width.

3. Precision (optional):

The precision is specified by a decimal point followed by an integer constant which defines:

- The maximum number of characters to be written in an 's' conversion
- The number of digits to appear after the decimal point in an 'e', 'E' or 'f' conversion
- The maximum number of significant digits for a 'g' or 'G' conversion
- The minimum number of digits to appear in a 'd', 'o', 'u', 'x' or 'X' conversion.

If the integer constant is replaced by an asterisk ('*'), an `int` argument supplies the precision. If the integer constant is omitted the value is taken to be zero.

4. Type specifier (optional):

   h Specifies that a following 'd', 'i', 'o', 'u', 'x' or 'X' conversion applies to a `short int` or `unsigned short int`, or a following 'n' conversion applies to a pointer to a `short int`.

   l Specifies that a following 'd', 'i', 'o', 'u', 'x' or 'X' conversion applies to a `long int` or `unsigned long int`, or a following 'n' conversion applies to a pointer to a `long int`.

   L Specifies that a following 'e', 'E', 'f', 'g' or 'G' conversion applies to a `long double`.

5. A single conversion character:

`d,i`	The int argument is converted to signed decimal format.
`o`	The int argument is converted to unsigned octal format.
`u`	The int argument is converted to unsigned decimal format.
`x`	The int argument is converted to unsigned hexadecimal format, using the letters 'a' to 'f'.
`X`	The int argument is converted to unsigned hexadecimal format, using the letters 'A' to 'F'.
`f`	The double argument is converted to the decimal format [–] xxx.xxxx. The number of characters after the decimal point is equal to the precision. The default precision is six.
`e, E`	The double argument is converted to the decimal format x.xxxxe±xx. The exponent is introduced with the conversion character. The number of characters after the decimal point is equal to the precision. The default precision is six.
`g, G`	The double argument is converted to an 'f' format if the exponent is less than −4 or greater than the precision. Otherwise 'g' is equivalent to 'e', and 'G' is equivalent to 'E'. Trailing zeros are removed from the result.
`c`	The int argument is written as a single character.
`s`	Characters are written from the string pointed to by the argument, up to the string terminating character.
`p`	The argument must be a pointer to a void and is converted to hex. format for printing.
`n`	The number of characters written so far will be put into the integer pointed to by the argument.
`%`	The % character is written.

Example:

```
#include <stdio.h>

int main( void )
{
   int i = 99;
   int count = 0;
   double fp = 1.5e5;
   char *s = "a sequence of characters";
   char nl = '\n';
   FILE *stream;

   if ( (stream = fopen("data.dat", "w")) == NULL)
      printf("Error opening data.dat for write\n");
   else
   {
     count+ = fprintf(stream,
                      "This is %s%c", s, nl);
     count+ = fprintf(stream,
                      "%d\n%f\n", i, fp);
     printf("Number of characters written to file
             was: %d\n", count);
   }
}
```

See also:

`fscanf` `printf`

`fputc` Writes a character to a file stream.

Synopsis:

```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

Arguments:

`int c`	The character to be written.
`FILE *stream`	A pointer to a file stream.

Results:

Returns the character written if successful. If a write error occurs, `fputc` returns `EOF` and sets the error indicator for the stream.

Errors:

`fputc` returns `EOF` if a write error occurs.

Description:

`fputc` converts `c` to an unsigned char, writes it to the output stream pointed to by `stream`, and moves the read/write position for the file stream as appropriate.

`fputc` is not included in the reduced library.

See also:

`fgetc` `putc`

`fputs` Writes a string to a file stream.

Synopsis:

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
```

Arguments:

`const char *s`	A pointer to the string to be written.
`FILE *stream`	A pointer to a file stream.

Results:

Returns non-negative if successful, and `EOF` if unsuccessful.

Errors:

`fputs` returns `EOF` if unsuccessful.

Description:

`fputs` writes the string pointed to by `s` to the file stream `stream`. The write does not include the string terminating character.

`fputs` is not included in the reduced library.

See also:

`fputc`

fread Reads records from a file.

Synopsis:

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb
             FILE *stream);
```

Arguments:

`void *ptr`	A pointer to a buffer that the records are read into.
`size_t size`	The size of an individual record.
`size_t nmemb`	The maximum number of records to be read.
`FILE *stream`	A pointer to a file stream.

Results:

Returns the number of records read. This may be less than `nmemb` if an error or end-of-file occurs. `fread` returns zero if `size` or `nmemb` is zero.

Errors:

None.

Description:

`fread` reads `nmemb` records of length `size` from the file stream `stream` into the array pointed to by `ptr`. The read/write file pointer is incremented by the number of characters read.

`fread` is not included in the reduced library.

Example:

```
#include <stdio.h>

FILE *stream;

int main()
{
   int i;
   int numout, numin;
   int buffin[10], buffout[10];
   FILE *stream;

   /* Write 10 integers to the file data.dat */
```

```
   stream = fopen("data.dat", "wb");
   if (stream == NULL)
      printf("error\);
   else
   {
      for (i = 0; i < 10; ++i)
        buffout[i] = i * i;

        /* Put values in buff */

      numout = fwrite((char *)buffout,
                       sizeof(int), 10, stream);
      printf(
      "number of integers written = %d\n", numout);
   }
   fclose(stream);

   /* Read 10 integers from the file data.dat */
   stream = fopen("data.dat", "rb");
   if (stream == NULL)
      printf("Error opening data.dat for binary
              write\n");
   else
   {
      numin = fread((char *)buffin,
                    sizeof(int), 10, stream);
      printf("number of integers read = %d\n", numin);
      for (i = 0; i < 10; ++i)
         printf("int %d is %d\n", i, buffin[i]);
   }
   fclose(stream};
}
```

See also:

`feof ferror fwrite`

`free` Frees an area of memory.

Synopsis:

```
#include <stdlib.h>
void free(void *ptr);
```

Arguments:

`void *ptr` A pointer to the area of memory to be freed.

Results:

Returns no result.

Errors:

If `ptr` does not match any of the pointers previously returned by `calloc`, `malloc`, or `realloc`, or if the space has already been freed by a call to `free` or `realloc`, a fatal runtime error occurs and the following message is displayed:

Fatal-C_Library-Error in free(), bad pointer or heap corrupted

Description:

`free` frees the area of memory pointed to by `ptr` if it has been previously allocated by `calloc`, `malloc`, or `realloc`. If `ptr` is a NULL pointer , no action occurs.

See also:

`calloc malloc realloc`

`free86` Frees host memory space allocated by `alloc86`. DOS only.

Synopsis:

```
#include <dos.h>
void free86(pcpointer p);
```

Arguments:

`pcpointer p` A pointer to the host memory block to be freed.

Results:

Returns no result.

Errors:

If an error occurs `free86` sets `errno` to the value `EDOS`. Any attempt to use `free86` on operating systems other than DOS also sets `errno` to `EDOS`. Failure of the function also generates the server error message:

```
[Encountered unknown primary tag (50)]
```

Description:

`free86` returns the block of host memory identified by `p` to DOS for re-use. `p` *must* be a `pcpointer` previously returned by `alloc86`.

`free86` is not included in the reduced library.

See also:

`alloc86`

freopen Closes an open file and reopens it in a given mode.

Synopsis:

```
#include <stdio.h>
FILE *freopen(const char *filename, const char
              *mode, FILE *stream);
```

Arguments:

`const char *filename`	The name of the file to be opened.
`const char *mode`	A string which specifies the mode in which the file is to be opened.
`FILE *stream`	A pointer to a file stream.

Results:

Returns the value of **stream** is associated with the newly opened file, or a NULL pointer if the file cannot be opened.

Errors:

If the open fails **freopen** returns a NULL pointer.

Description:

freopen attempts to close the file associated with the file stream **stream**. Failure to close the file is ignored, error and end-of-file indicators for the stream are cleared, and **freopen** then opens the file referenced by **filename** and associates the file with the file stream **stream**.

The file is opened in the mode specified by the string **mode**. Valid modes are the same as for **fopen**.

freopen is not included in the reduced library.

freopen is normally used for redirecting the **stdin**, **stdout** and **stderr** streams.

Example:

```
#include <stdio.h>

int main( void )
{
   FILE *stream;

   /* assign stdout to a named file */
   printf("This text goes to stdout\n");

   stream = freopen("data.dat", "w", stdout);
   if (stream == NULL)
      printf("Couldn't freopen stdout to
              data.dat\n");
   else
   {
      printf("This text goes to data.dat\n");
      fclose(stream);
   }
}
```

See also:

`fopen`

frexp

Separates a floating point number into a mantissa and an integral power of 2.

Synopsis:

```
#include <math.h>
double frexp(double value, int *exp);
```

Arguments:

`double value`	The floating point number.
`int *exp`	A pointer to an integer where the exponent is stored.

Results:

Returns the mantissa part of value. The mantissa is returned in the range [0.5 ... 1) or zero. The exponent is stored in the `int` pointed to by `exp`.

Errors:

A domain error may occur.

Description:

`frexp` separates the floating point number value into a mantissa and an integral power of 2. The exponent is stored in the `int` pointed to by `exp`. The mantissa is returned by the function.

If x is the value returned by `frexp` and y is the exponent stored in `*exp` then:

```
value = x * 2**y
```

If `value` is zero then both `x` and `y` will be zero.

Example:

```
#include <math.h>
#include <stdio.h>

int main( void )
{
   double x;
   double mantissa;
   int exponent;
```

```
        x = 3.141;
        mantissa = frexp(x,&exponent);
        printf("x = %f, mantissa = %f, exponent = %d\n",
                x, mantissa, exponent);
}
/*
 *   Output:
 *
 *           x = 3.141000, mantissa = 0.785250,
 *               exponent = 2
 *
 */
```

See also:

`ldexp frexpf`

frexpf

Separates a floating point number into a mantissa and an integral power of 2.

Synopsis:

```
#include <mathf.h>
float frexpf(float value, int *exp);
```

Arguments:

`float value`	The floating point number.
`int *exp`	A pointer to the int into which the exponent is put.

Results:

Returns the mantissa part of value. The mantissa is returned in the range [0.5...1) or zero. The exponent is stored in the `int` pointed to by `exp`.

Errors:

None.

Description:

`float` form of `frexp`.

See also:

`ldexpf` `frexp`

`from86` Transfers host memory to the transputer. DOS only.

Synopsis:

```
#include <dos.h>
int from86(int len, pcpointer there, char *here);
```

Arguments:

`int len`	The number of bytes of host memory to be transferred.
`pcpointer there`	A pointer to the host memory block.
`char *here`	A pointer to the receiving block in transputer memory.

Results:

Returns the actual number of bytes transferred.

Errors:

Returns the number of bytes transferred until the error occurred and sets `errno` to the value `EDOS`. Any attempt to use `from86` on systems other than DOS also sets `errno` to `EDOS`. Failure of the function also generates the server error message:

```
[Encountered unknown primary tag (50)]
```

Description:

`from86` transfers `len` bytes of host memory starting at `there` to a corresponding block starting at `here` in transputer memory. The function returns the number of bytes actually transferred. The host memory block used will normally have been previously allocated by a call to `alloc86`.

`from86` is not included in the reduced library.

See also:

`to86 alloc86`

`fscanf` Reads formatted input from a file stream.

Synopsis:

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, ...);
```

Arguments:

`FILE *stream`	An input file stream.
`const char *format`	A format string.
`...`	Subsequent arguments to the format string.

Results:

Returns the number of inputs which have been successfully converted. If an end-of-file character occurred before any conversions took place, `fscanf` returns `EOF`.

Errors:

If an end-of-file character occurred before any conversions took place, `fscanf` returns `EOF`. Other failures cause termination of the procedure.

Description:

`fscanf` matches the data read from the input stream `stream` to the specifications set out by the format string. The format string can include white space, ordinary characters, or conversion tokens:

1. Whitespace causes the next series of white space characters read to be ignored.

2. Ordinary characters in the format string cause the characters read to be compared to the corresponding character in the format string. If the characters do not match, conversion is terminated.

3. A conversion token in the format string causes the data sequence read in to be checked to see if it is in the specified format. If it is, it is converted and placed in the appropriate argument following the format string. If the data is not in the correct format, conversion is terminated.

The conversion tokens consist of the following items:

1. Token signifier:

`%` (percent character)

2. Assignment suppressor (optional):

   `*` (asterisk). This causes the data sequence to be read in but not assigned to an argument. Tokens that use the assignment suppressor should not have a corresponding argument in the argument list.

3. Maximum width (optional):

   The width is a decimal integer constant defining the maximum number of characters to be read.

4. Type specifier (optional):

   `h` Specifies that a following 'd', 'i', 'n', 'o', 'u', or 'x' conversion applies to a `short int` or `unsigned short int`.
   `l` Specifies that a following 'd', 'i', 'n', 'o', 'u' or 'x' conversion applies to a `long int` or `unsigned long int`, and a following 'e', 'f' or 'g' conversion applies to a `double`.
   `L` Specifies that a following 'e', 'f' or 'g' conversion applies to a `long double`.

5. A single conversion character:

   `d` Expects an (optionally signed) decimal integer. Requires a pointer to an integer as the corresponding argument.
   `i` Expects an (optionally signed) integer constant. Requires a pointer to an integer as the corresponding argument.
   `o` Expects an (optionally signed) octal integer. Requires a pointer to an integer as the corresponding argument.
   `u` Expects an (optionally signed) decimal integer. Requires a pointer to an unsigned integer as the corresponding argument.
   `x` Expects an (optionally signed) hex integer (optionally preceded by an 0x or 0X). Requires a pointer to an integer as the corresponding argument.

e, f, g Expects an (optionally signed) floating point character consisting of the following sequence of characters:

1. A plus or minus sign (optional).

2. A sequence of decimal digits, which may contain a decimal point.

3. An exponent (optional) consisting of an 'E' or 'e' followed by an optional sign and a string of decimal digits. Requires a pointer to a `double` as the corresponding argument.

s Expects a string. Requires a pointer to an array large enough to hold (size of the string plus a terminating null char) characters as the corresponding argument.

[Signifies the start of a scanset.

[set] Expects a string made up of the characters included between the square brackets.

[^ set] expects a string made up of characters which are not included between the square brackets. The right bracket character can be included in the match set by beginning the scan set as follows: [ ] or [^].

[- set] Treated as any other character, no matter where it appears in the scan set.

Requires a pointer to an array large enough to hold the size of the string plus a terminating null character, (which will be added automatically) as the corresponding argument.

p Expects a hexadecimal string. Requires a pointer to a void pointer as the corresponding argument.

n The number of characters received so far will be put into the integer pointed to by the argument. This does not increment the assignment count returned.

% Matches the % character.

Any mismatch between the token format and the data received causes an early termination of `fscanf`.

`fscanf` is not included in the reduced library.

Example:

```
#include <stdio.h>

int main( void )
{
```

```
    FILE *stream;
    int numin;
    int numout;
    float fp;
    int i;

    /* Create a file containing a number of items */
    stream = fopen("data.dat", "w");

    if (stream == NULL)
       printf("Couldn't open data.dat for write\n");
    else
    {
       numout = fprintf(stream, "%f %d",
                 3.141, 1024);
       printf(
              "Number of characters written: %d\n",
               numout);
    }

    fclose(stream);

    /* Read a number of items from the file */
    stream = fopen("data.dat", "r");
    if (stream == NULL)
       printf("Couldn't open data.dat for read\n");
    else
    {
       numin = 0;
       numin = numin + fscanf(stream, "%f", &fp);
       numin = numin + fscanf(stream, "%d", &i);
       printf("Number of fields read: %d\n", numin);
       printf("Items read were: %f %d\n",fp, i);
    }
}

/*   Output:
 *
 *          Number of characters written: 13
 *          Number of fields read: 2
 *          Items read were: 3.141000, 1024
 */
```

See also:

fprintf

fseek Sets the file pointer to a specified offset.

Synopsis:

```
#include <stdio.h>
int fseek(FILE *stream, long int offset,
          int whence);
```

Arguments:

`FILE *stream`	A pointer to a file stream.
`long int offset`	The distance the read/write pointer is moved.
`int whence`	The start position for the read/write pointer.

Results:

Returns non-zero if called incorrectly, otherwise **fseek** returns zero.

Errors:

fseek returns non-zero on error.

Description:

fseek is used to move the read/write position pointer of a file to a specified offset within the file stream **stream**. The offset is measured from a position defined by **whence** and can take the following values:

1 SEEK_SET is the start of the file stream.
2 SEEK_CUR is the current position in the file stream.
3 SEEK_END is the end of the file stream.

If the file stream is a text stream the offset should either be zero or **whence** should be set to SEEK_SET, and **offset** should be a value returned by a **ftell**.

fseek clears the end-of-file indicator for **stream** and undoes the effects of **ungetc**. The file stream may be both read from and written to after **fseek** has been called, provided the stream has been opened in an appropriate mode.

Example:

```
#include <stdio.h>
```

```
int main( void )
{
   FILE *stream;
   int result;
   stream = fopen("data.dat", "wb+");

   if (stream == NULL)
      printf("couldn't open data.dat for write\n");
   else
   {
      fprintf(stream,"%s","123456789");

      /* Reset to beginning of file */
      result = fseek(stream, 0L, SEEK_SET);

      if (result)
         printf("couldn't do fseek\n");
      else
         printf("first char in file is: %c\n",
                getc(stream));

      /* Reset to beginning of file */
      result = fseek(stream, 0L, SEEK_SET);

      /* Move to third byte in file */
      result = fseek(stream, 2L, SEEK_CUR);

      if (result)
         printf("couldn't do fseek\n");
      else
         printf("third char in file is: %c\n",
                getc(stream));

      /* Move to last byte in file */
      result = fseek(stream, -1L, SEEK_END);

      if (result)
         printf("couldn't do fseek\n");
      else
         printf("last char in file is: %c\n",
                getc(stream));
   }
}
```

See also:

fsetpos, ftell, ungetc

fsetpos

Sets the read/write file pointer to an `fpos_t` value obtained from `fgetpos`.

Synopsis:

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

Arguments:

`FILE *stream`	A pointer to a file stream.
`const fpos_t *pos`	A pointer to an object containing the new value of the read/write file pointer.

Results:

Returns zero if the operation was successful, and non-zero on failure.

Errors:

If the operation was unsuccessful, `fsetpos` sets `errno` to `EFILPOS` and returns a non-zero value.

Description:

`fsetpos` sets the read/write position pointer of the file stream `stream` to the value in `pos`. `pos` shall contain a value previously returned by `fgetpos`.

A successful call to `fsetpos` clears the end-of-file indicator for the stream and will undo the effects of an `ungetc` operation on the same stream. The file stream may be both read from and written to after `fsetpos` has been called, provided it has been opened in an appropriate mode.

`fsetpos` is not included in the reduced library.

```
#include <stdio.h>

int main( void )
{
   FILE *stream;
   fpos_t filepos;
   int ch;

   stream = fopen("data.dat","w+");
   if (stream == NULL)
     printf("Couldn't open data.dat for read\n");
```

```
   else
   {
      fprintf(stream, "123456789");
      rewind(stream);
      ch = getc(stream);
      printf("First char in file is '%c'\n",ch);

      /*
       * Remember: getc() advances file pointer,
       *           so it now points
       * to the second character in the file.
       */

      if (fgetpos(stream,&filepos) != 0)
        printf("Error with fgetpos\n");

      ch = getc(stream);
      printf("Second char in file is '%c'\n",ch);
      ch = getc(stream);
      printf("Third character in file is '%c'\n",ch);

      if (fsetpos(stream,&filepos) !=0)
        printf("Error with fsetpos\n");

      ch = getc(stream);
      printf(
      "Reset file ptr and read 2nd char which is '%c'\n",
       ch);
      fclose(stream);
    }
}
```

See also:

`fgetpos fseek ungetc`

`ftell` Returns the position of the read/write pointer in a file stream.

Synopsis:

```
#include <stdio.h>
long int ftell(FILE *stream);
```

Arguments:

`FILE *stream` A pointer to a file stream.

Results:

Returns the current value of the read/write position indicator for the file stream `stream`, or −1 on error.

Errors:

`ftell` returns −1 on error and sets `errno` to `EFILPOS`.

Description:

`ftell` returns the current value of the read/write position indicator for the file stream `stream`. For a binary stream the value is the number of characters from the beginning of the file. For a text stream the value is unspecified but can be used by `fseek` to reposition the file position indicator to its original position at the time of the call to `ftell`.

`ftell` is not included in the reduced library.

See also:

`fseek`

`fwrite` Writes records from an array into a file.

Synopsis:

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

Arguments:

`void *ptr`	A pointer to a buffer that the records are read from.
`size_t size`	The size of an individual record.
`size_t nmemb`	The maximum number of records to be written.
`FILE *stream`	A pointer to a file stream.

Results:

Returns the number of records written. This may be less than `nmemb` if a write error occurs.

Errors:

`fwrite` returns zero if `size` or `nmemb` is zero.

Description:

`fwrite` writes `nmemb` records of length `size` from the array pointed to by `ptr` into the file stream `stream`. The read/write file pointer is incremented by the number of characters written. If an error occurs, the value of the file position indicator is indeterminate.

`fwrite` is not included in the reduced library.

See also:

`fread`

`get_param` Reads parameters for a configured process.

Synopsis:

```
#include<misc.h>
void *get_param(int n);
```

Arguments:

`int n` The index of the required parameter in the interface list.

Results:

Returns no result.

Errors:

The function returns NULL on error. Possible errors are:

1 Using the function when it is not valid, i.e. from a program not configured using `icconf`.

2 Using a value of `n` less than 1.

3 Using a value of `n` which is greater than the number of available parameters.

Description:

`get_param` reads parameters from the list specified in the `interface` attribute for a configured process. It can only be used from a program which has been configured using `icconf` and has *not* been linked with the entry points `MAIN.ENTRY`, `PROC.ENTRY` or `PROC.ENTRY.RC` (used only for compatibility with code generated by previous toolsets, as described in appendix F *'occam interface code'* of the accompanying User Manual).

`get_param` is used to access the parameters given to a process in the interface list at configuration level. It returns the `n`th parameter in the parameter list (n is a non-zero positive integer). If the parameter is a scalar then a pointer to the parameter is returned. If the parameter a channel or array then the channel or array pointer itself is returned.

The following example shows how a C program can use `get_param` to obtain the value of a variable defined in the interface parameter list of a process defined at configuration level. The configuration description includes all the placements necessary to configure the process on a single processor.

C program:

```
#include <stdio.h>
#include <stdlib.h>
#include <misc.h>

int main ()
{
    int *value;

    value = (int *)get_param(3);
    printf("value = %d\n", *value);
    exit_terminate(EXIT_SUCCESS);
}
```

Configuration description:

```
/* Hardware description */
T414(memory = 2M) B403;

connect B403.link[0], host;

/* Software description */
process(stacksize = 20k, heapsize = 20k,
        interface(input in,
                  output out,
                  int value)) test;

test(value = 427);

input from_host;
output to_host;

connect test.in, from_host;
connect test.out, to_host;

/* Network mapping */
use "test1.lku" for test;
place test on B403;

place to_host   on host;
place from_host on host;

place test.in  on B403.link[0];
place test.out on B403.link[0];
```

The C program obtains the value 427 by reading the third interface parameter to the configured process `test` and then displays it.

`getc` Gets a character from a file.

Synopsis:

```
#include <stdio.h>
int getc(FILE *stream);
```

Arguments:

`FILE *stream` A pointer to a file stream.

Results:

Returns the next character from the file stream.

Errors:

If the next character is the end-of-file character, or a read error occurs, `getc` returns EOF.

Description:

`getc` returns the next character from the opened file identified by the file stream pointer, and advances the read/write position indicator for the file stream.

`getc` is not included in the reduced library.

See also:

`fgetc getchar putc`

`getenv` Returns the name of a host environment variable.

Synopsis:

```
#include <stdlib.h>
char *getenv(const char *name);
```

Arguments:

`const char *name`	A pointer to the host variable name to be matched.

Results:

Returns a pointer to the matched string in the host environment variable list. If no match is found, a NULL pointer is returned.

Errors:

None.

Description:

`getenv` returns the string associated with the host environment variable *name*. The string must not be modified by the program but can be overwritten by a subsequent call to `getenv`.

`getenv` is not included in the reduced library.

Note: Care should be taken when calling `getenv` in a concurrent environment. Calls to the function by independently executing, unsynchronised processes may corrupt the returned char pointer.

Example:

```
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   char *envvar;
   envvar = getenv("IBOARDSIZE");
   if (envvar == NULL)
      printf("IBOARDSIZE variable not set\n");
   else
      printf("IBOARDSIZE is : %\n",envvar);
}
```

`getkey` Reads a character from the keyboard.

Synopsis:

```
#include <iocntrl.h>
int getkey(void);
```

Arguments:

None.

Results: Returns the ASCII value of the character, or −1 on error.

Errors: Returns −1 if an error occurs.

Description: `getkey` returns the value of the next character typed at the keyboard. The routine waits indefinitely for the next keystroke and only returns when a key is available. The effect on any buffered data in the standard input stream is host-defined. The character read is not echoed at the terminal.

`getkey` is not included in the reduced library.

See also:

`pollkey`

gmtime Returns a UTC time.

Synopsis:

```
#include <time.h>
struct tm *gmtime(const time_t *timer);
```

Arguments:

`const time_t *timer`	Calendar time pointed to by `timer`.

Results:

Returns a pointer to a broken-down time expressed as UTC time, or NULL if UTC time is unavailable.

Errors:

Returns NULL if UTC time is not available.

Description:

`gmtime` converts a calendar time into a standard time format. The standard format used is Coordinated Universal Time (UTC).

Note: UTC is unavailable in this implementation and `gmtime` *always* returns NULL.

See also:

`asctime` `ctime` `difftime` `localtime` `strftime` `clock` `mktime` `time`

host_info Gets data about the host system.

Synopsis:

```
#include <host.h>
void host_info(int *host, int *os, int *board);
```

Arguments:

`int *host`	A pointer to an int where the host type code will be stored.
`int *os`	A pointer to an int where the operating system type code will be stored.
`int *board`	A pointer to an int where the board type code will be stored.

Results: Returns no result. Writes host system attributes into `host`, `os`, and `board`.

Errors: If any host attribute is unavailable it is given the value 0.

Description: `host_info` returns information about the host environment. It stores codes for the host type, host operating system and transputer board in the locations pointed to by `host`, `os`, and `board` respectively.

`host_info` is not included in the reduced library.

The values that `host` can take are defined in the header `host.h` and are as follows:

1 `_IMS_HOST_PC`

2 `_IMS_HOST_NEC`

3 `_IMS_HOST_VAX`

4 `_IMS_HOST_SUN3`

5 `_IMS_HOST_SUN4`

6 `_IMS_HOST_SUN386i`

7 `_IMS_HOST_APOLLO`

The values that `os` can take are as follows:

1 `_IMS_OS_DOS`

2 `_IMS_OS_HELIOS`

3 `_IMS_OS_VMS`

4 `_IMS_OS_SUNOS`

5 `_IMS_OS_CMS`

The values that `board` can take are as follows:

1 `_IMS_BOARD_B004`

2 `_IMS_BOARD_B008`

3 `_IMS_BOARD_B010`

4 `_IMS_BOARD_B011`

5 `_IMS_BOARD_B014`

6 `_IMS_BOARD_DRX11`

7 `_IMS_BOARD_QT0`

8 `_IMS_BOARD_B015`

9 `_IMS_BOARD_CAT`

10 `_IMS_BOARD_B016`

11 `_IMS_BOARD_UDP_LINK`

int86 Performs a DOS software interrupt. DOS only.

Synopsis:

```
#include <dos.h>
int int86(int intno, union REGS *inregs,
          union REGS *outregs);
```

Arguments:

`int intno`	The host software interrupt ID.
`union REGS *inregs`	Values to be placed in processor registers.
`union REGS *outregs`	Register values after the interrupt.

Results:

Returns the value of the **ax** register after the interrupt.

Errors:

Returns zero (0) on error and sets `errno` to the value `EDOS`. Any attempt to use `int86` on operating systems other than DOS also sets `errno` to `EDOS`. Failure of the function also generates the server error message:

```
[Encountered unknown primary tag (50)]
```

Description:

`int86` calls the host software interrupt identified by `intno` with the registers set to `inregs`. Register values after the interrupt are returned in `outregs` and the contents of the **ax** register is returned as the function result.

Segment registers **cs**, **ds**, **ex**, and **ss** are not set.

`int86` is not included in the reduced library.

See also:

`int86x intdos`

`int86x` Software interrupt with segment register setting. DOS only.

Synopsis:

```
#include <dos.h>
int int86x(int intno, union REGS *inregs,
           union REGS *outregs,
           struct SREGS *segregs);
```

Arguments:

`int intno`	The DOS software interrupt ID.
`union REGS *inregs`	Values to be placed in processor registers.
`union REGS *outregs`	Register values after the interrupt.
`struct SREGS *segregs`	Values to be placed in segment registers.

Results:

Returns the value of the **ax** register after the interrupt.

Errors:

Returns zero (0) on error and sets `errno` to the value `EDOS`. Any attempt to use `int86x` on operating systems other than DOS also sets `errno` to `EDOS`. Failure of the function also generates the server error message:

```
[Encountered unknown primary tag (50)]
```

Description:

`int86x` calls the host software interrupt identified by `intno` with the registers set to `inregs` and the segment registers set to `segregs`. Register values after the interrupt are returned in `outregs` and the contents of the **ax** register is returned as the function result.

`int86x` is useful for DOS calls which take pointers to objects, normally specified by combining a 16-bit register with a segment register. If only some of the segment registers are modified, `segread` should be used to read values from the others. Failure to do so can produce unpredictable results.

See also:

`int86 intdosx`

`intdos` Performs a DOS interrupt. DOS only.

Synopsis:

```
#include <dos.h>
int intdos(union REGS *inregs,
           union REGS *outregs);
```

Arguments:

`union REGS *inregs`	Values to be placed in processor registers.
`union REGS *outregs`	Register values after the interrupt.

Results:

Returns the value of the **ax** register after the interrupt.

Errors:

Returns zero (0) on error and sets `errno` to the value `EDOS`. Any attempt to use `intdos` on operating systems other than DOS also sets `errno` to `EDOS`. Failure of the function also generates the server error message:

```
[Encountered unknown primary tag (50)]
```

Description:

As `int86` but calls the specific host software interrupt identified by hexadecimal 21 (DOS function call).

See also:

`int86` `intdosx`

intdosx DOS interrupt with segment register setting. DOS only.

Synopsis:

```
#include <dos.h>
int intdosx(union REGS *inregs,
            union REGS *outregs,
            struct SREGS *segregs);
```

Arguments:

`union REGS *inregs`	Values to be placed in processor registers.
`union REGS *outregs`	Register values after the interrupt.
`struct SREGS *segregs`	Values to be placed in segment registers.

Results:

Returns the value of the **ax** register after the interrupt.

Errors:

Returns zero (0) on error and sets `errno` to the value `EDOS`. Any attempt to use `intdosx` on operating systems other than DOS also sets `errno` to `EDOS`. Failure of the function also generates the server error message:

```
[Encountered unknown primary tag (50)]
```

Description:

As `intdos` but also sets segment registers.

See also:

`intdos int86x`

`isalnum` Tests whether a character is alphanumeric.

Synopsis:

```
#include <ctype.h>
int isalnum(int c);
```

Arguments:

`int c` The character to be tested.

Results:

Returns non-zero (true) if the character is alphanumeric and zero (false) if it is not.

Errors:

None.

Description:

`isalnum` tests whether the character `c` is in one of the following sets of alphabetic and numeric characters:

'a' to 'z' 'A' to 'Z' '0' to '9'

`isalnum` is implemented both as a macro and a function.

See also:

`isalpha isdigit`

`isalpha` Tests whether a character is alphabetic.

Synopsis:

```
#include <ctype.h>
int isalpha(int c);
```

Arguments:

`int c` The character to be tested.

Results:

Returns non-zero (true) if the character is alphabetic and zero (false) if it is not.

Errors:

None.

Description:

`isalpha` tests whether `c` is in one of the following sets of alphabetic characters:
'a' to 'z' 'A' to 'Z'

`isalpha` is implemented both as a macro and a function.

See also:

`isalnum` `isdigit`

`isatty` Tests for a standard stream.

Synopsis:

```
#include <iocntrl.h>
int isatty(int fd);
```

Arguments:

`int fd` A file descriptor.

Results:

Returns 1 (true) if the file descriptor refers to a standard stream, otherwise returns 0 (false).

Errors:

None.

Description:

`isatty` determines whether a given file descriptor refers to one of the default terminal files `stdin`, `stdout`, and `stderr`.

`isattty` is not included in the reduced library.

`iscntrl` Tests whether a character is a control character.

Synopsis:

```
#include <ctype.h>
int iscntrl(int c);
```

Arguments:

`int c` The character to be tested.

Results:

Returns non-zero (true) if the character is a control character and zero (false) if it is not.

Errors:

None.

Description:

`iscntrl` determines whether `c` is a control character (ASCII codes 0–31 and 127).

`iscnrtl` is implemented both as a macro and a function.

`isdigit` Tests whether a character is a decimal digit.

Synopsis:

```
#include <ctype.h>
int isdigit(int c);
```

Arguments:

`int c` The character to be tested.

Results:

Returns non-zero (true) if the character is a digit and zero (false) if it is not.

Errors:

None.

Description:

`isdigit` tests whether `c` is one of the following decimal digit characters:

'0' '1' '2' '3' '4' '5' '6' '7' '8' '9'

`isdigit` is implemented both as a macro and a function.

See also:

`isalnum` `isalpha`

`isgraph` Tests whether a character is printable (non-space).

Synopsis:

```
#include <ctype.h>
int isgraph(int c);
```

Arguments:

`int c` The character to be tested.

Results:

Returns non-zero (true) if the character is a printable character (other than space) and zero (false) if it is not.

Errors:

None.

Description:

`isgraph` tests whether `c` belongs to the set of printable characters excluding the space character (' '). The space character is considered in this test to be non-printable.

`isgraph` is implemented both as a macro and a function.

See also:

`iscntrl` `isprint` `isspace`

`islower` Tests whether a character is a lower-case letter.

Synopsis:

```
#include <ctype.h>
int islower(int c);
```

Arguments:

`int c` The character to be tested.

Results:

Returns non-zero (true) if the character is a lower-case letter and zero (false) if it is not.

Errors:

None.

Description:

`islower` tests whether `c` is a character in the set of lower case characters:

'a' to 'z'

`islower` is implemented both as a macro and a function.

See also:

`isupper`

`isprint` Tests whether a character is printable (includes space).

Synopsis:

```
#include <ctype.h>
int isprint(int c);
```

Arguments:

`int c` The character to be tested.

Results:

Returns non-zero (true) if the character is printable and zero (false) if it is not.

Errors:

None.

Description:

`isprint` tests whether `c` is a printable character (ASCII character codes 32–126).

Note: Unlike `isgraph`, `isprint` considers the space character (' ') to be printable.

`isprint` is implemented both as a macro and a function.

See also:

`isgraph`

`ispunct` Tests to see if a character is a punctuation character.

Synopsis:

```
#include <ctype.h>
int ispunct(int c);
```

Arguments:

`int c` The character to be examined.

Results:

Returns non-zero (true) if the character is a punctuation character and zero (false) if it is not.

Errors:

None.

Description:

`ispunct` tests whether `c` is a punctuation character. For the purposes of this test a punctuation is any printable character other than an alphanumeric or space (' ') character.

`ispunct` is implemented both as a macro and a function.

See also:

`iscntrl isgraph isprint`

`isspace` Tests to see if a character is one which affects spacing.

Synopsis:

```
#include <ctype.h>
int isspace(int c);
```

Arguments:

`int c` The character to be tested.

Results:

Returns non-zero (true) if the character is a space character and zero (false) if it is not.

Errors:

None.

Description:

`isspace` tests whether `c` belongs to the set of characters which produce white space. Characters which generate white space are as follows:

TAB (escape sequence '`\t`')	SPACE (' ')
LINE FEED/NEWLINE ('`\n`')	Vertical TAB ('`\v`')
FORM FEED ('`\f`')	RETURN ('`\r`').

`isspace` is implemented both as a macro and a function.

`isxdigit` Tests to see if a character is a hexadecimal digit.

Synopsis:

```
#include <ctype.h>
int isxdigit(int c);
```

Arguments:

`int c` The character to be tested.

Results: Returns non-zero (true) if the character is a hexadecimal digit and zero (false) if it is not.

Errors:

None.

Description: `isxdigit` tests whether `c` belongs to the set of hexadecimal digits. These are as follows:

'a' 'b' 'c' 'd' 'e' 'f' 'A' 'B' 'C' 'D' 'E' 'F' '0' '1' '2' '3' '4' '5' '6' '7' '8' '9'

`isxdigit` is implemented both as a macro and a function.

`labs` Calculates the absolute value of a long integer.

Synopsis:

```
#include <stdlib.h>
long int labs(long int j);
```

Arguments:

`long int j` A long integer.

Results:

Returns the absolute value of `j` as a `long int`.

Errors:

If the result cannot be represented the behaviour of `labs` is undefined.

Description:

`labs` calculates the absolute value of the long int `j`.

See also:

`abs`

`ldexp` Multiplies a floating point number by an integer power of two.

Synopsis:

```
#include <math.h>
double ldexp(double x, int exp);
```

Arguments:

`double x`	The floating point number.
`int exp`	The exponent.

Results:

Returns the value of: `x * 2 * *exp`.

Errors:

A range error will occur if the result of `ldexp` would cause overflow or underflow. In this case `errno` is set to `ERANGE`.

Description:

`ldexp` calculates the value of : `x * 2 * *exp`.

See also:

`frexp`

`ldexpf` Multiplies a `float` number by an integral power of two.

Synopsis:

```
#include <mathf.h>
float ldexpf(float x, int exp);
```

Arguments:

`float x`	The floating point number.
`int exp`	The exponent.

Results:

Returns the value of: x * 2**exp

Errors:

A range error will occur if the result of `ldexpf` would cause overflow or underflow. In this case `errno` is set to `ERANGE`.

Description:

`float` form of `ldexp`.

See also:

`ldexp` `frexp`

`ldiv` Calculates the quotient and remainder of a long division.

Synopsis:

```
#include <stdlib.h>
ldiv_t ldiv(long int numer, long int denom);
```

Arguments:

`long int numer`	The numerator.
`long int denom`	The denominator.

Results:

Returns a structure of type `ldiv_t` which consists of the quotient and remainder. The structure contains:

`long int quot`	The quotient.
`long int rem`	The remainder.

Errors:

If the result cannot be represented the behaviour of `ldiv` is undefined.

Description:

`ldiv` calculates the quotient and remainder formed by dividing the numerator `num` by the denominator `denom`. All values are of type `long int`.

See also:

`div`

`localeconv` Gets numeric formatting data in the current locale.

Synopsis:

```
#include <locale.h>
struct lconv *localeconv(void);
```

Arguments:

None.

Results:

Returns a pointer to a structure of type `lconv` which defines components of the current locale.

Errors:

None.

Description:

The components of the `lconv` structure are set according to the current locale (defined in `locale.h`), and a pointer to this structure is returned. Previous values in `lconv` are overwritten.

The `lconv` structure should not be overwritten by the program but may be altered by a call to `setlocale`.

ANSI C supports only the standard "C" locale.

See also:

`setlocale`

`localtime` Converts the local time into a `tm` structure format.

Synopsis:

```
#include <time.h>
struct tm *localtime(const time_t *timer);
```

Arguments:

`const time_t *timer` A pointer to a location containing a time.

Results:

Returns a pointer to a `tm` calendar structure, containing the value of the timer in a specific format.

Errors:

None.

Description:

`localtime` is used to convert a time stored in the value pointed to by `timer` to the `tm` structure format.

Example:

```
/*
  Prints the current date and time in a
                          default format
*/

#include <time.h>
#include <stdio.h>

int main()
{
 time_t current;
 struct tm *calendar;

 time(&current);
 calendar = localtime(&current);
 printf ("\n
Date and time = %s\n",
          asctime(calendar));
}
```

Note: Care should be taken when calling `localtime` in a concurrent environment. Calls to the function by independently executing, unsynchronised processes may corrupt the returned time value.

See also:

`asctime ctime strftime clock difftime mktime time`

`log` Calculates the natural logarithm of the `double` argument.

Synopsis:

```
#include <math.h>
double log(double x);
```

Arguments:

`double x` A number.

Results:

Returns the natural log of `x`.

Errors:

A domain error occurs if `x` is negative. In this case `errno` is set to `EDOM`.

A range error occurs if `x` is zero. In this case `log` returns the value `HUGE_VAL` (with the same sign as the correct value of the function) and `errno` is set to `ERANGE`.

Description:

`log` calculates the natural (base e) logarithm of a number.

See also:

`log10 logf`

`logf` Calculates the natural logarithm of a `float` number.

Synopsis:

```
#include <mathf.h>
float logf(float x);
```

Arguments:

`float x` A number.

Results:

Returns the natural log of `x`.

Errors:

A domain error occurs if `x` is negative. In this case `errno` is set to `EDOM`.

A range error occurs if `x` is zero. In this case `logf` returns the value `HUGE_VAL_F` (with the same sign as the correct value of the function) and `errno` is set to `ERANGE`.

Description:

`float` form of `log`.

See also:

`log log10f`

`log10` Calculates the base-10 logarithm of the `double` argument.

Synopsis:

```
#include <math.h>
double log10(double x);
```

Arguments:

`double x` A number.

Results:

Returns the base ten log of `x`.

Errors:

A domain error occurs if `x` is negative. In this case `errno` is set to `EDOM`.

A range error occurs if `x` is zero. In this case `log10` returns the value `HUGE_VAL` (with the same sign as the correct value of the function) and `errno` is set to `ERANGE`.

Description:

`log10` calculates the base 10 logarithm of a number.

See also:

`log log10f`

`log10f` Calculates the base-10 logarithm of a `float` number.

Synopsis:

```
#include <mathf.h>
float log10f(float x);
```

Arguments:

`float x` A number.

Results:

Returns the base ten log of `x`.

Errors:

A domain error occurs if `x` is negative. In this case `errno` is set to `EDOM`.

A range error occurs if `x` is zero. In this case `log10f` returns the value `HUGE_VAL_F` (with the same sign as the correct value of the function) and `errno` is set to `ERANGE`.

Description:

`float` form of `log10`.

See also:

`log10` `logf`

longjmp

Performs a non-local jump to the given environment.

Synopsis:

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

Arguments:

`jmp_buf env`	An array holding the environment to be restored.
`int val`	The value to be returned by `longjmp`.

Results:

When `longjmp` returns, the effect is as if the corresponding `setjmp` had returned the value of `val`. If `val` is zero, `setjmp` returns 1 (this is because `setjmp` is only allowed to return zero the first time it is called).

Errors:

None.

Description:

`longjmp` performs a non-local jump to the environment saved in `env`, by a previous call to `setjmp`. It returns in such a way that, to the program, it appears that the `setjmp` function has returned the value `val`.

Example:

```
#include <setjmp.h>
#include <stdio.h>
#include <stdlib.h>

jmp_buf env1;

int sub_function()
{
  /* .....
     ..... */

    longjmp(env1, 3);
}

int main()
{
```

```
int a;

switch(a=setjmp(env1))
{
 case 0:  printf("1st time in top level\n");
          break;
 default: printf("longjmp to top level
                - code %d\n", a);
          exit( EXIT\_\/\/SUCCESS );
}
sub_function();
}
```

See also:

setjmp

`lseek` Repositions a file pointer.

Synopsis:

```
#include <iocntrl.h>
int lseek(int fd, long int offset, int origin);
```

Arguments:

`int fd`	A file descriptor.
`long int offset`	The offset by which the file position will move.
`int origin`	The start position for the seek.

Results:

Returns the new file position, or −1 on error.

Errors:

If an error occurs lseek sets `errno` to the value `EIO`.

Description:

`lseek` moves the current position within the file with file descriptor `fd`. The offset is measured from a position specified by `origin`:

`L_SET`	The start of the file.
`L_INCR`	The current position in the file.
`L_CUR`	The end of the file.

`lseek` is not included in the reduced library.

`malloc` Allocates a specified area of memory.

Synopsis:

```
#include <stdlib.h>
void *malloc(size_t size);
```

Arguments:

`size_t size` The size of the space to be allocated in bytes.

Results:

Returns a pointer to the allocated space if the allocation was successful. Otherwise a null pointer is returned.If size is zero `malloc` returns a NULL pointer.

Errors:

If there is not enough free space a null pointer is returned.

Description:

`malloc` allocates an area of memory of `size` bytes. The allocated space is not initialised.

Example:

```
/* Allocate 500 bytes pointed to by array1 */

char *array1;

array1 = (char *)malloc(500);
```

See also:

`calloc free realloc`

`max_stack_usage` Calculates runtime stack usage.

Synopsis:

```
#include <misc.h>
long max_stack_usage(void);
```

Arguments:

None.

Results:

Returns the number of bytes of stack space used by the program.

Errors:

If stack checking is not enabled in the compiler the function returns zero.

Description:

`max_stack_usage` returns the approximate number of stack bytes used by the program up to the point where the function is called. A leeway of 150 words is included in the returned value to account for library usage, in which there is no stack checking.

Note: This function can only be used when stack checking is enabled. If stack checking is disabled the function returns 0 (zero).

memchr Finds first occurrence of a character in an area of memory.

Synopsis:

```
#include <string.h>
void *memchr(const void *s, int c, size_t n);
```

Arguments:

`const void *s`	A pointer to the area of memory to be searched.
`int c`	The character to be searched for.
`size_t n`	The size of the area of memory to be searched.

Results:

If the character is found, `memchr` returns a pointer to the matched character. It returns a null pointer if the character `c` is not in the first `n` characters of the area of memory.

Errors:

None.

Description:

`memchr` finds the first occurrence of `c` in the first `n` characters of the area of memory pointed to by `s`. `c` is converted to an `unsigned char` before the search begins.

Example:

```
char buffer[100];
char *pointer_to_p;

/*
  Find the first occurrence of "p"
  in the buffer
*/

pointer_to_p = memchr(buffer, 'p', 100);
```

See also:

`strchr`

memcmp Compares characters in two areas of memory.

Synopsis:

```
#include <string.h>
int memcmp(const void *s1, const void *s2,
           size_t n);
```

Arguments:

`const void *s1`	A pointer to one of the areas of memory to be compared.
`const void *s2`	A pointer to the other area of memory to be compared.
`size_t n`	The number of characters to be compared.

Results:

Returns the following:

> A negative integer if the `s1` area of memory is numerically less than the `s2` area of memory.
>
> A zero value if the two areas of memory are numerically the same.
>
> A positive integer if the `s1` area of memory is numerically greater than the `s2` area of memory.

Errors:

None.

Description:

`memcmp` compares the first `n` characters of the areas of memory pointed to by `s1` and `s2`.

The comparison is of the numerical values of the ASCII characters.

See also:

`strcmp`

memcpy

Copies characters from one area of memory to another (no memory overlap allowed).

Synopsis:

```
#include <string.h>
void *memcpy(void *s1, const void *s2, size_t n);
```

Arguments:

`void *s1`	A pointer to the destination of the copy.
`const void *s2`	A pointer to the source of the copy.
`size_t n`	The number of characters to be copied.

Results:

Returns the unchanged value of `s1`.

Errors:

The behaviour of `memcpy` is undefined if the source and destination overlap.

Description:

`memcpy` copies `n` characters from the area of memory pointed to by `s2` (the source) to the area of memory pointed to by `s1` (the destination). The behaviour of `memcpy` is undefined if the source and target areas overlap.

```
char source[200];
destination[200];

memcpy(destination, source, 200);
```

Calls to `memcpy` can be replaced by the compiler predefine `_memcpy` by redefining the function name. `_memcpy` is implemented directly as transputer assembly code in selected cases. For details see section 11.4 in the accompanying User Manual.

See also:

`memmove` `_memcpy`

memmove

Copies characters from one area of memory to another.

Synopsis:

```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```

Arguments:

`void *s1`	A pointer to the destination of the copy.
`const void *s2`	A pointer to the source of the copy.
`size_t n`	The number of characters to be copied.

Results:

Returns the unchanged value of `s1`.

Errors:

None.

Description:

`memmove` copies `n` characters from the area of memory pointed to by `s2` (the source) to the area of memory pointed to by `s1` (the destination). The copying is carried out even if the areas of memory overlap.

See also:

`memcpy`

`memset` Fills a given area of memory with the same character.

Synopsis:

```
#include <string.h>
void *memset(void *s, int c, size_t n);
```

Arguments:

`void *s`	A pointer to the area of memory to be filled.
`int c`	The character to be used for filling.
`size_t n`	The number of characters in the area of memory be filled.

Results:

Returns the unchanged value of `s`.

Errors:

None.

Description:

`memset` fills the first `n` characters of the area of memory pointed to by `s` with the value of the character `c`. `c` is converted to an `unsigned char` before the filling takes place.

Example:

```
/*
  Zero the first hundred bytes of a buffer
*/

char buffer[200];

memset(buffer,'\0', 100);
```

mktime Converts a `tm` structure into a `time_t` value.

Synopsis:

```
#include <time.h>
time_t mktime(struct tm *timeptr);
```

Arguments:

`struct tm *timeptr` A structure containing a calendar time.

Results:

Returns the value of `timeptr` as a number of seconds.

Errors:

If the time in `timeptr` cannot be represented as a `time_t` type, `mktime` returns −1, cast to `time_t`.

Description:

`mktime` converts the values given in the `tm` structure pointed to by `timeptr` into a time of type `time_t`. The values of the structure components `tm_wday` and `tm_yday` are ignored, all elements in `tm` are set to appropriate values, and the time value `time_t` represented by the `tm` structure is returned.

Values processed by `mktime` from the structure `timeptr` are not restricted to the ranges specified on page 24. Values outside the specified ranges are converted automatically by `mktime` to produce a valid `time_t` value.

Example:

```
#include <time.h>
#include <stdio.h>

/* Initialise broken_down_time,
        omitting weekday etc */

int main( void )
{
   struct tm broken_down_time = {0, 0, 11, 2, 0,
                                 93, 0, 0, 0};

   time_t cal_time;
   cal_time = mktime(&broken_down_time);
```

```
        printf("Weekday is %d\n",
               broken_down_time.tm_wday);
    }
```

See also:

`asctime ctime localtime clock difftime time`

`modf` Splits a `double` number into fractional and integral parts.

Synopsis:

```
#include <math.h>
double modf(double value, double *intptr);
```

Arguments:

`double value`	The number to be split.
`double *intptr`	A pointer to the recipient of the integral part.

Results:

Returns the fractional part of `value` (the integral part is stored in `*intptr`).

Errors:

None.

Description:

`modf` splits `value` into a fractional and integral part. Each part has the same sign as `value`. The integral part is stored in `*intptr` and the fractional part is returned by `modf`.

See also:

`modff`

`modff` Splits the float argument into fractional and integral parts.

Synopsis:

```
#include <mathf.h>
float modff(float value, float *intptr);
```

Arguments:

`float value`	The number to be split.
`float *intptr`	A pointer to the recipient of the integral part.

Results:

Returns the fractional part of `value` (the integral part is stored in `*intptr`).

Errors:

None.

Description:

`float` form of `modf`.

See also:

`modf`

open

Opens a file stream. File handling primitive.

Synopsis:

```
#include <iocntrl.h>
int open(char *name, int flags);
```

Arguments:

`char *name`	The name of the file to be opened.
`int flags`	Bit values which specify the mode in which the file is to be opened.

Results:

Returns a file descriptor for the file opened or -1 on error.

Errors:

If an error occurs `errno` is set to `EIO`.

Description:

`open` opens the low level file `name` in a mode specified by `flags`. `open` is the low level file function used by `fopen`.

`open` is not included in the reduced library.

The `flags` parameter is a combination of bit values joined using the 'bitwise or' (|) operator. The bit values that can be specified are as follows:

Read/write Modes:

Flag	Meaning
O_RDONLY	Read only mode (priority 3).
O_WRONLY	Write only mode (priority 2).
O_RDWR	Read/write mode (priority 1).

File creation modes:

Flag	Meaning
O_APPEND	Characters appended to file (priority 1).
O_TRUNC	File truncated before writing (priority 2).

File Types:

Flag	Meaning
O_BINARY	File opened in binary mode (priority 2).
O_TEXT	File opened as a text file. (priority 1).

The **flags** parameter should combine values from each of the three sections above. For example, to open a binary file for writing in append mode the call would be as follows:

```
open(filename, O_BINARY | O_WRONLY | O_APPEND);
```

To avoid conflicts between the various combinations of modes, the flag values are assigned priority levels and are decoded accordingly. Priority increases with increasing number. For example, if both `O_WRONLY` (priority 2) and `O_RDONLY` (priority 3) are specified in the same call `O_WRONLY` is ignored.

Priority levels also imply a default setting for `open`, namely: Read only/Text mode (`O_RDONLY | O_TEXT`). (File create modes have no significance on a read only file).

If a file which already exists is opened using `O_TRUNC` (open for writing in truncate mode), and if the host system permits it, the file will be overwritten.

See also:

creat

`perror` Writes an error message to standard error.

Synopsis:

```
#include <stdio.h>
void perror(const char *s);
```

Arguments:

`const char *s` A pointer to an error message string.

Results:

No value is returned.

Errors:

None.

Description:

`perror` writes the string `s` to the standard error output, followed by a colon, space, and the error message represented by the value in `errno`. The entire message is followed by a newline.

Message strings are the same as those returned by `strerror` given the argument `errno`.

`perror` is not included in the reduced library.

See also:

`strerror`

`pollkey` Gets a character from the keyboard.

Synopsis:

```
#include <iocntrl.h>
int pollkey(void);
```

Arguments:

None.

Results:

`pollkey` returns the ASCII value of a key pressed on the keyboard. It immediately returns with −1 if no keystroke is available.

Errors:

None.

Description:

`pollkey` gets a single character from the keyboard. If no keystroke is available the routine returns immediately with −1. The effect on any buffered data in the standard input stream is host-defined. The character read from the keyboard is not echoed at the terminal.

`pollkey` is not included in the reduced library.

See also:

`getkey`

pow Calculates x to the power y.

Synopsis:

```
#include <math.h>
double pow(double x, double y);
```

Arguments:

`double x`	A number.
`double y`	The exponent.

Results:

Returns the value of `x` to the power `y`.

Errors:

A domain error will occur in the following situations:

1.	x == 0	AND	y <= 0
2.	x < 0	AND	y is not an integer

In these cases `errno` is set to `EDOM`.

A range error will occur if the result of `pow` is too large to fit in a double. In this case `pow` returns the value `HUGE_VAL` (with the same sign as the correct value of the function) and `errno` is set to `ERANGE`.

Description:

`pow` calculates the value of `x` raised to the power `y`.

See also:

`powf`

`powf` Calculates x to the power of y where both x and y are floats.

Synopsis:

```
#include <mathf.h>
float powf(float x, float y);
```

Arguments:

`float x`	A number.
`float y`	The exponent.

Results:

Returns the value of a number to the power `y`.

Errors:

A domain error will occur in the following situations:

1.	x == 0	AND	y <= 0
2.	x < 0	AND	y is not an integer

In these cases `errno` is set to `EDOM`.

A range error will occur if the result of `powf` is too large to fit in a double. In this case `powf` returns the value `HUGE_VAL_F` (with the same sign as the correct value of the function) and `errno` is set to `ERANGE`.

Description:

`float` form of `pow`.

See also:

`pow`

`printf` Writes a formatted string to standard output.

Synopsis:

```
#include <stdio.h>
int printf(const char *format, ...);
```

Arguments:

`const char *format`	A format string.
`...`	Subsequent arguments to the format string.

Results:

Returns the number of characters written, or a negative value if an output error occurred.

Errors:

`printf` returns a negative value if an output error occurred.

Description:

`printf` writes the string pointed to by `format` to standard output. When `printf` encounters a percent sign % in the string, it expands the equivalent argument into the format defined by the format tokens after the %. The meaning of the format string is as described for `fprintf`.

`printf` is not included in the reduced library.

See also:

`fprintf`

`ProcAfter` Blocks a process until a specified time.

Synopsis:

```
#include <process.h>
void ProcAfter(int time);
```

Arguments:

`int time` The time at which the process will restart.

Results:

Returns no result.

Errors:

None.

Description:

Delays execution of the current process until a specified time. Time is expressed as an integer clock value.

See also:

`ProcWait`

`ProcAlloc` Allocates process space and initialises its structure.

Synopsis:

```
#include <process.h>
Process *ProcAlloc(void (*func)(),
                   int sp, int nparam, ...);
```

Arguments:

`*func`	A pointer to the function to be created as a parallel process.
`int sp`	The amount of stack space required for the process. `sp` must be specified in bytes.
`int nparam`	The number of parameters to the process (if all parameters are word-sized), or the number of words taken up by the parameters.
`...`	A list of word-sized parameters to the process.

Results:

Returns a pointer to the process structure, or a NULL pointer if the allocation is unsuccessful.

Errors:

Returns NULL if the allocation is unsuccessful.

Description:

`ProcAlloc` allocates memory space for a process and initialises the allocated structure.

Note: All processes *must* be allocated (by a call to `ProcAlloc` or `ProcInit`) before use.

`ProcAlloc` takes as parameters a pointer to a function which is to be spawned as a process, the size of workspace required by the process, and parameters to the function. It returns a pointer to an initialised process structure describing the process. The pointer is used to start the process by passing it to one of the process execution functions.

If `sp` is specified as zero, stack sizes of 4Kbytes for 32-bit transputers and 1Kbyte for 16-bit transputers is used.

`nparam` specifies the number of *words* required on the stack initially for the

function's parameters. If parameters are all word-sized (after default promotions have taken place) then **nparam** should equal the number of parameters in the list. If parameters are not all word-sized then **nparam** must be the same as the number of words occupied. For example, if a structure is passed that occupies four words, and all other parameters are word-sized, then **nparam** must be increased by four.

ProcAlloc must have as its first parameter a pointer to a process structure. **nparam** must not include this process pointer.

Note: When using parameters larger than one word, allowance must be made for any default type promotions performed by the compiler by rounding up aggregate types to the nearest word.

float variables cannot be passed directly as parameters because the promotion is to type **double**. In this case, and in all others where the parameter is larger than a word, pointers should be used.

ProcAlloc uses **malloc** to allocate stack space (allocated from the heap). If the call to **malloc** is unsuccessful, **ProcAlloc** returns a NULL pointer. All calls to **ProcAlloc** should be followed by a check for successful allocation and secure handling of a NULL result. The consequences of running an unallocated process are undefined.

ProcAlloc calls the lower level function **ProcInit** to initialise the process structure.

Example:

```
/* To set up fred as a concurrent process
   with default workspace */

#include <process.h>

void fred(Process *p, int a, int b, int c)
{
  /* code for fred */
}

Process *p;

p = ProcAlloc(fred, 0, 3, 1, 2, 3);

if (p == NULL)
   abort();

/* p is a process structure for fred. Actual
```

```
        parameters for the process will be:
        a = 1; b = 2; c = 3.    */
```

See also:

`ProcInit malloc`

`ProcAllocClean` Frees space allocated by `ProcAlloc`.

Synopsis:

```
#include <process.h>
void ProcAllocClean(Process *p);
```

Arguments:

`Process *p`	A pointer to a process structure.

Results:

None.

Errors:

If an invalid pointer is passed to `ProcAllocClean` a fatal runtime error occurs and the following message is displayed:

Fatal-C_Library-Bad pointer to process clean function

and the processor is halted. If the reduced library is used no message is displayed.

Description:

`ProcAllocClean` is used to clean up after a process when it is known to have terminated. The process is denoted by the process pointer passed in as the argument and must have been initially set up using `ProcAlloc`. It will *not* work correctly for processes set up using `ProcInit` and if used in such a case may produce undefined behaviour.

`ProcAllocClean` removes the process structure pointed to by its argument from the list of initialised processes and frees any heap space used for the process structure and workspace.

Caution: `ProcAllocClean` can only be used with synchronous processes, i.e. those started using `ProcPar` or `ProcParList`, and can be safely used only after the call to `ProcPar` or `ProcParList` has returned. Any other use of this function may give rise to undefined behaviour.

See also:

`ProcAlloc` `ProcInitClean`

ProcAlt

Waits for input from multiple processes.

Synopsis:

```
#include <process.h>
int ProcAlt(Channel *c1, ...);
```

Arguments:

`Channel *c1`	The first in a NULL terminated list of pointers to channels.
`...`	The remainder of the list.

Results:

Returns an index into the parameter list for the ready channel.

Errors:

None.

Description:

`ProcAlt` blocks the calling process until one of the channel parameters is ready to input. The index returned for the ready channel is an integer which indicates the position of the channel in the parameter list. The index numbers begin at zero for the first parameter.

`ProcAlt` only returns when a channel is ready to input. It does not perform the input operation, which must be done by the code following the call to `ProcAlt`.

Example:

```
/* select from channels c1, c2, c3 */

#include <process.h>

Channel *c1, *c2, *c3;
int i;

/* allocate all channels */

i = ProcAlt(c1, c2, c3, NULL);
switch(i)
{
  case 0: /* c1 selected */
          /* consume input from c1 */
```

```
                 break;
        case 1: /* c2 selected */
                /* consume input from c2 */
                break;
        case 2: /* c3 selected */
                /* consume input from c3 */
                break;
    }
```

See also:

`ProcAltList`

`ProcAltList` Waits for inputs from a list of processes.

Synopsis:

```
#include <process.h>
int ProcAltList(Channel **clist);
```

Arguments:

`Channel **clist`	An array of pointers to channels termintted by NULL.

Results:

Returns an index into the `clist` array for the ready channel, or −1 if the first element in the array is NULL (the array is empty).

Errors:

Returns −1 if `clist` is empty.

Description:

As `ProcAlt` but takes an array of pointers to channels. Returns −1 if the `clist` array is empty.

See also:

`ProcAlt`

ProcGetPriority Returns the priority of the process.

Synopsis:

```
#include <process.h>
int ProcGetPriority(void);
```

Arguments:

None.

Results:

Returns zero (0) for a high priority process and one (1) for a low priority process.

Errors:

None.

Description:

Determines the priority level (high or low) of the process from which it is called. The macros `PROC_HIGH` and `PROC_LOW` are defined for use with this function.

See also:

`ProcReschedule`

ProcInit Initialises a process.

Synopsis:

```
#include <process.h>
int ProcInit(Process *p, void (*func)(), int *ws,
             int wssize, int nparam, ...);
```

Arguments:

`Process *p`	A pointer to a process structure.
`int *func`	A pointer to the function to be expressed as a process.
`int *ws`	A pointer to the stack space to be used.
`int wssize`	The size of the stack space. `wssize` must be specified in bytes.
`int nparam`	The number of parameters to the process (if all parameters are word-sized), or the number of words taken up by the parameters.
`...`	A list of word sized parameters to the process.

Results:

Returns zero (0) if successful, non-zero otherwise.

Errors:

If insufficient space has been allocated for parameters to the function, the routine returns a non-zero value. If the workspace pointed by `ws` has not been been allocated from the *heap*, a fatal runtime error occurs and the following message is displayed:

Fatal-C_Library-Incorrect allocation of process workspace

Description:

`ProcInit()` takes as input a pointer to an existing `Process` structure and a pointer to the stack space to be used, and initializes the process structure and workspace for the function according to its workspace and parameter space requirements. `ProcInit()` is called by `ProcAlloc()` to initialise the process structure.

As with `ProcAlloc`, `nparam` specifies the number of *words* required on the stack initially for the function's parameters. If parameters are all word-sized (after default promotions have taken place) then `nparam` should equal the number of parameters in the list. If parameters are not all word-sized then `nparam` must

be the same as the number of words occupied. For example, if a structure is passed that occupies four words, and all other parameters are word-sized, then `nparam` must be increased by four.

Note: When using parameters that consist of more than one word, take care to allow for any default type promotions performed by the compiler, and be sure to round up aggregate types to the nearest word.

`float` variables cannot be passed directly as parameters because the promotion is to type `double`. In this case, and in all others where the parameter is larger than a word, pointers should be used.

`ProcInit` checks that enough space has been allocated for the function parameters, and that space has been allocated from the *heap*.

Example:

```
/* To set fred up as a concurrent process
   with 4k of stackspace
*/

#include <process.h>
#include <stdlib.h>
#define SIZE 4096

void fred(Process *p, int a, int b, int c)
{
  /* code for fred */
}

/* code fragment */

Process *p;
char *ws;

p = (Process *)malloc(sizeof(Process));

/* check whether p is NULL */

ws = (int*)malloc(SIZE);

/* check whether ws is NULL */

if (ProcInit(p, fred, ws, SIZE, 3, 1, 2, 3))
{
  /* error */
}
```

```
/* p is a process structure for fred.
   When the process is started the parameters
   will be: a = 1; b = 2; c = 3.  */
```

See also:

`ProcAlloc`

`ProcInitClean` Frees space allocated by `ProcInit`.

Synopsis:

```
#include <process.h>
void ProcInitClean(Process *p);
```

Arguments:

`Process *p`	A pointer to a process structure.

Results:

None.

Errors:

If an invalid pointer is passed to `ProcInitClean` a fatal runtime error occurs and the following message is displayed:

Fatal-C_Library-Bad pointer to process clean function

and the processor is halted. If the reduced library is used no message is displayed.

Description:

`ProcInitClean` is used to clean up after a process when it is known to have terminated. The process is denoted by the process pointer passed in as the argument and must have been initially set up using `ProcInit`. It will *not* work correctly for processes set up using `ProcAlloc` and if used in such a case may produce undefined results.

`ProcInitClean` removes the process structure pointed to by its argument from the list of initialised processes. After `ProcInitClean` has been called the memory space allocated for the process structure and workspace may be safely freed. If this space is freed before a call to `ProcInitClean` then the behaviour is undefined. Note that `ProcInitClean` does not itself free the *workspace*, which must be performed by the programmer.

Caution: `ProcInitClean` can only be used with synchronous processes, i.e. those started using `ProcPar` or `ProcParList`, and can be safely used only after the call to `ProcParList` or `ProcPar` has returned. Any other use of this function may give rise to undefined behaviour.

See also: `ProcInit` `ProcAllocClean`

ProcPar Starts a group of processes in parallel.

Synopsis:

```
#include <process.h>
void ProcPar(Process *p1, ...);
```

Arguments:

`Process *p1`	The first in a list of pointers to process structures.
`...`	The remainder of the list. Terminated by NULL.

Results:

Returns no result.

Errors:

None.

Description:

`ProcPar` takes a NULL terminated list of pointers to processes and starts them in parallel with each other at the priority of the calling process. Control is returned to the calling process when all the processes in the list terminate. The process pointers are either returned from `ProcAlloc` or are pointers to existing processes initialised by `ProcInit`.

`ProcParam` should be used before the process is executed. If it is used while the process is running the results may be unpredictable.

```
/* start the four processes denoted by process
   pointers p1, p2, p2, p4 in parallel. */

#include <process.h>

Process *p1, *p2, *p3, *p4;

/* Set up and allocate processes */

ProcPar(p1, p2, p3, p4, NULL);
```

See also:

`ProcParList`

`ProcParam` Changes process parameters.

Synopsis:

```
#include <process.h>
void ProcParam(Process *p, ...);
```

Arguments:

`Process *p`	A pointer to a process structure.
`...`	A list of parameters to the process.

Results:

Returns no result.

Errors:

None.

Description:

`ProcParam` alters parameters in an already allocated process. The number of parameters specified should be the same as the number required by the process. Any extra parameters given are ignored. If fewer than the required number are specified the unspecified parameters remain undefined.

The process pointers are either returned from `ProcAlloc`, or are pointers to existing processes initialised by `ProcInit`.

Example:

```
/* p is the process pointer for a function
   which takes three parameters */

Process *p;

ProcParam(p, 1, 2, 3);

/* This call to ProcParam sets the parameters of
   the process associated with p to 1, 2, 3. */
```

See also:

`ProcAlloc`

ProcParList Starts a group of parallel processes.

Synopsis:

```
#include <process.h>
void ProcParList(Process **plist);
```

Arguments:

`Process **plist`	A array of pointers to processes terminated by NULL.

Results:

Returns no result.

Errors:

None.

Description:

As `ProcPar` but takes an array of pointers to processes. The pointers are either returned directly from `ProcAlloc` or are pointers to processes initialised by `ProcInit`.

See also:

`Procpar`

`ProcPriPar` Starts a pair of processes at high and low priority.

Synopsis:

```
#include <process.h>
void ProcPriPar(Process *phigh, Process *plow)
```

Arguments:

`Process *phigh`	A pointer to the high priority process.
`Process *plow`	A pointer to the low priority process.

Results:

Returns no result.

Errors:

Any attempt to call `ProcPriPar` from a high priority process generates a run-time fatal error and the following message is displayed:

```
Fatal-C_Library-Nested Pri Pars are illegal
```

Description:

Starts two processes in parallel, one at high priority and one at low priority. Process pointers will have been returned directly from `ProcAlloc`, or are pointers to processes initialised by `ProcInit`.

`ProcPriPar` cannot be called from a high priority process.

See also:

`ProcPar`

ProcReschedule Reschedules a process.

Synopsis:

```
#include <process.h>
void ProcReschedule(void);
```

Arguments:

None.

Results:

Returns no result.

Errors:

None.

Description:

Causes the current process to be rescheduled, that is, placed at the end of the active process queue.

See also:

`ProcGetPriority`

`ProcRun` Starts a process at the current priority.

Synopsis:

```
#include <process.h>
void ProcRun(Process *p);
```

Arguments:

`Process *p` A pointer to a process.

Results:

Returns no result.

Errors:

None.

Description:

Executes a process in parallel with the calling process and at the same priority. The two processes run independently and any interaction between them must be specifically set up using channel communication routines. The process pointer is returned directly from `ProcAlloc` or is a pointer to a process initialised by `ProcInit`.

Care should be taken that unsynchronised processes do not attempt to communicate with the server when it has been terminated by the main program. Synchronising channels can be used to guard against this. For more details see section 4.7.4 in the accompanying User Manual.

See also:

`ProcRunHigh ProcRunLow ProcPar ProcParList ProcPriPar`

`ProcRunHigh` Starts a high priority process.

Synopsis:

```
#include <process.h>
void ProcRunHigh(Process *p);
```

Arguments:

`Process *p`	A pointer to a process.

Results:

Returns no result.

Errors:

None.

Description:

As `ProcRun` but starts the process at high priority. Process pointers will have been returned directly from `ProcAlloc`, or are pointers to processes initialised by `ProcInit`.

As with `ProcRun` care should be taken that processes started with this function terminate before the main program.

See also:

`ProcRun ProcRunLow ProcPar ProcParList ProcPriPar`

ProcRunLow Starts a low priority process.

Synopsis:

```
#include <process.h>
void ProcRunLow(Process *p);
```

Arguments:

`Process *p` A pointer to a process.

Results:

Returns no result.

Errors:

None.

Description:

As **ProcRun** but starts the process at low priority. As with **ProcRun** care should be taken that processes started with this function terminate before the main program.

See also:

`ProcRunHigh ProcRun ProcPar ProcParList ProcPriPar`

ProcSkipAlt Checks specified channels for ready input.

Synopsis:

```
#include <process.h>
int ProcSkipAlt(Channel *c1, ...);
```

Arguments:

`Channel *c1`	The first in a list of pointers to channels.
`...`	The remainder of the list. Terminated by NULL.

Results:

Returns an index into the parameter list for the channel ready to input, or −1 if no channel is ready.

Errors:

None.

Description:

As `ProcAlt` but does not wait for a ready channel. If no channel is ready `ProcSkipAlt` returns immediately with the value −1.

Example:

```
/* select from channels c1, c2, c3 */

#include <process.h>

Channel *c1, *c2, *c3;
int i;

/* set up channels */

i = ProcSkipAlt(c1, c2, c3, NULL);
switch(i)
{
  case -1: /* no channel ready */
  case 0:  /* c1 selected */
           /* consume input from c1 */
           break;
  case 1:  /* c2 selected */
```

```
                        /* consume input from c2 */
                        break;
            case 2:     /* c3 selected */
                        /* consume input from c3 */
                        break;
        }
```

See also:

`ProcAlt` `ProcSkipAltList`

ProcSkipAltList Checks a list of channels for ready input.

Synopsis:

```
#include <process.h>
int ProcSkipAltList(Channel **clist);
```

Arguments:

`Channel **clist`	An array of pointers to channels terminated by NULL.

Results:

As `ProcSkipAlt`.

Errors:

None.

Description:

As `ProcSkipAlt` but takes a list of pointers to channels.

See also:

`ProcSkipAlt`

`ProcStop` Deschedules a process.

Synopsis:

```
#include <process.h>
void ProcStop(void);
```

Arguments:

None.

Results:

Returns no result.

Errors:

None.

Description:

Stops the current process.

`ProcTime` Determines the transputer clock time.

Synopsis:

```
#include <process.h>
int ProcTime();
```

Arguments:

None.

Results:

Returns the value of the transputer clock.

Errors:

None.

Description:

Determines the transputer clock time. The value of the high priority clock is returned for high priority processes and the value of the low priority clock is returned for low priority processes. Values returned by this function can be used by `ProcTimeAfter`, `ProcTimePlus`, and `ProcTimeMinus`.

See also:

`ProcTimeAfter` `ProcTimePlus` `ProcTimeMinus`

ProcTimeAfter Determines relationship of clock values.

Synopsis:

```
#include <process.h>
int ProcTimeAfter(const int time1, const int time2);
```

Arguments:

`int time1`	A transputer clock value returned by `ProcTime`.
`int time2`	A transputer clock value returned by `ProcTime`.

Results:

Returns 1 if `time1` is after `time2`, otherwise 0.

Errors:

None.

Description:

Determines the relationship between two transputer clock values. Remember that the transputer clock is cyclic.

See also:

`ProcTime` `ProcTimePlus` `ProcTimeMinus`

ProcTimeMinus Subtracts two transputer clock values.

Synopsis:

```
#include <process.h>
int ProcTimeMinus(const int time1, const int time2);
```

Arguments:

`int time1`	A transputer clock value returned by `ProcTime`.
`int time2`	A transputer clock value returned by `ProcTime`.

Results:

Returns the result of subtracting `time2` from `time1`.

Errors:

None.

Description:

Subtracts one clock value from another using modulo arithmetic. No overflow checking takes place and the clock values are cyclic.

See also:

`ProcTime ProcTimeAfter ProcTimeMinus`

ProcTimePlus Adds two transputer clock values.

Synopsis:

```
#include <process.h>
int ProcTimePlus(const int time1, const int time2);
```

Arguments:

`time1/time2`	Clock values returned by `ProcTime`.

Results:

Returns the result of adding `time1` to `time2`.

Errors:

None.

Description:

Adds one clock value to another using modulo arithmetic. No overflow checking takes place and the values are cyclic.

See also:

`ProcTime` `ProcTimeAfter` `ProcTimeMinus`

ProcTimerAlt Checks input channels or times out.

Synopsis:

```
#include <process.h>
int ProcTimerAlt(int time, Channel *c1, ...);
```

Arguments:

`int time`	The time after which the function aborts if no communication occurs. Represented by a specific clock value.
`Channel *c1`	The first in a list of pointers to channels.
`...`	The remainder of the list. The list must be terminated by NULL.

Results:

Returns an index to the parameter list, or −1 if the routine times out.

Errors:

None.

Description:

As `ProcAlt` but controlled by a timeout. If `time` is exceeded before any communication occurs the routine terminates and returns the value −1.

Example:

```
/* select from channels c1, c2, c3 */

#include <process.h>

Channel *c1, *c2, *c3;
int i;

/* set up channels */

i = ProcTimerAlt(ProcTimePlus(ProcTime(), 50000),
                 c1, c2, c3, NULL);
switch(i)
{
  case -1: /* timed out */
  case 0:  /* c1 selected */
           /* consume input from c1 */;
```

```
                    break;
        case 1:     /* c2 selected */
                    /* consume input from c2 */
                    break;
        case 2:     /* c3 selected */
                    /* consume input from c3 */
                    break;
      }
```

See also:

`ProcAlt` `ProcTimerAltList`

`ProcTimerAltList` Checks a list of channels or times out.

Synopsis:

```
#include <process.h>
int ProcTimerAltList(int time, Channel **clist)
```

Arguments:

`int time`	The time after which the function aborts if no communication occurs. Represented by a specific clock value.
`Channel **clist`	An array of pointers to channels terminated by NULL.

Results:

Returns no result.

Errors:

None.

Description:

As `ProcTimerAlt`, but takes an array of pointers to channels.

See also:

`ProcTimerAlt`

`ProcWait` Suspends a process for a specified time.

Synopsis:

```
#include <process.h>
void ProcWait(int time);
```

Arguments:

`int time`	The time delay measured in transputer clock ticks.

Results:

Returns no result.

Errors:

None.

Description:

Suspends execution of a process for a specified period of time. When the period expires, the process starts.

See also:

`ProcAfter`

`putc` Writes a character to a file stream.

Synopsis:

```
#include <stdio.h>
int putc(int c, FILE *stream);
```

Arguments:

`int c`	The character to be written.
`FILE *stream`	A pointer to a file stream.

Results:

Returns the character written if the write is successful, or **`EOF`** if a write error occurs.

Errors:

`putc` returns **`EOF`** if a write error occurs.

Description:

`putc` converts `c` to an unsigned char, writes it to the output stream pointed to by **`stream`**, and advances the read/write position indicator for the file stream.

`putc` is not included in the reduced library.

See also:

`fputc`

`putchar` Writes a character to standard output.

Synopsis:

```
#include <stdio.h>
int putchar(int c);
```

Arguments:

`int c` The character to be written.

Results:

Returns the character written if successful. If a write error occurs, `putchar` returns `EOF`.

Errors:

`putchar` returns `EOF` if a write error occurs.

Description:

`putchar` converts `c` to an unsigned char, writes it to the standard output stream, and advances the read/write position indicator for the file stream.

`putchar` is not included in the reduced library.

See also:

`fputc` `getchar` `putc`

`puts` Writes a line to standard output.

Synopsis:

```
#include <stdio.h>
int puts(const char *s);
```

Arguments:

`const char *s` A pointer to the string to be written.

Results:

Returns non-negative if successful, EOF if unsuccessful.

Errors:

`puts` returns EOF if unsuccessful.

Description:

`puts` writes the string pointed to by `s` to the standard output file stream, followed by a newline character. The write does not include the string terminating character.

`puts` is not included in the reduced library.

See also:

`fputs getchar gets putchar`

qsort Sorts an array of objects.

Synopsis:

```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Arguments:

`void *base`	A pointer to the start of the array.
`size_t nmemb`	The number of objects in the array.
`size_t size`	The size of the array objects.
`int (*compar)(const void *, const void *)`	A pointer to the comparison function.

Results:

Returns no value.

Errors:

None.

Description:

`qsort` sorts objects in the array pointed to by `base` into ascending order, according to comparisons performed by the function pointed to by `compar`. The array contains `nmemb` objects of `size` bytes.

The comparison function must return an integer less than, equal to, or greater than zero, depending on whether the first argument to the function is considered to be less than, equal to, or greater than the second argument.

Example:

```
#include <stdio.h>
#include <stdlib.h>

int sort_compare(const void *arg1,
                 const void *arg2)
{
  return (int) (*((int *)arg1) - *((int *)arg2));
{

int main()
```

```
{
  int i[10] = {1, 4, 6, 5, 2, 7, 9, 3, 8, 0};
  int j;

  qsort(i, 10, sizeof(int), sort_compare);
  for (j = 0; j < 10; ++j)
    printf("%d\n", i[j]);
}
```

See also:

`bsearch`

`raise` Sends a signal to the executing program.

Synopsis:

```
#include <signal.h>
int raise(int sig);
```

Arguments:

`int sig`	A signal number, as defined in `signal.h`.

Results:

Returns zero (0) if successful, non-zero if unsuccessful.

Errors:

If `raise` is called with an unrecognised signal number, it returns a non-zero value.

Description:

`raise` is used to send a signal to the running program. The actual function called in response to a `raise` call depends on the function specified in `signal`.

Signals which can be raised are listed under the signal handling setup function `signal`.

See also:

`signal`

`rand` Generates a pseudo-random number.

Synopsis:

```
#include <stdlib.h>
int rand(void);
```

Arguments:

None.

Results:

Returns a positive pseudo-random integer.

Errors:

None.

Description:

`rand` generates a pseudo-random integer in the range 0 to `RAND_MAX`.

Note: Successive calls to the function by unsynchronised parallel processes will each produce a new number from the pseudo-random sequence.

See also:

`srand`

`read` Reads bytes from a file stream. File handling primitive.

Synopsis:

```
#include <iocntrl.h>
int read(int fd, char *buf, int n);
```

Arguments:

`int fd`	A file descriptor.
`char *buf`	A pointer to a buffer where the bytes will be stored.
`int n`	The maximum number of bytes that `read` will attempt to obtain.

Results:

Returns the number of bytes read or −1 on error.

Errors:

If an error occurs `read` sets `errno` to the value `EIO`.

Description:

`read` attempts to read `n` bytes from the file described by `fd` into the buffer pointed to by `buf`. `read` may return a value less than `n` if an end of file occurred. `n` may be zero or negative but in these cases no input will occur.

Note: Care should be taken when calling `localtime` in a concurrent environment. Calls to the function by independently executing, unsynchronised processes may change the return value.

`read` is not included in the reduced library.

See also:

`write`

`realloc` Changes the size of an object in memory.

Synopsis:

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

Arguments:

`void *ptr`	A pointer to the area of memory.
`size_t size`	The new size of the area of memory.

Results:

Returns a pointer to the allocated space. If it was not possible to allocate `size` bytes, or if the size requested is zero and the pointer parameter is NULL, `realloc` returns a NULL pointer.

Errors:

If it is not possible to allocate `size` bytes, `realloc` returns a NULL pointer. If `ptr` does not point to an area of memory which was previously allocated by `calloc`, `malloc`, or `realloc` and which has not been deallocated by a call to `free` or `realloc`, a fatal runtime error occurs and the following message is generated:

Fatal-C_Library-Error in realloc(), bad pointer or heap corrupted

Description:

`realloc` allocates an area of memory of `size` size, and copies the previously allocated area of memory pointed to by `ptr` into the newly allocated area. If the previous area is larger than the new area, the overflow will be lost.

If `ptr` is NULL, `realloc` behaves like a call to `malloc`.

If `size` is zero and `ptr` is not a NULL pointer, the object pointed to by `ptr` is freed. If `ptr` is invalid a runtime error from `free` may be generated.

See also:

`calloc free malloc`

remove Removes a file.

Synopsis:

```
#include <stdio.h>
int remove(const char *filename);
```

Arguments:

`const char *filename`	A pointer to the filename string.

Results:

Returns zero (0) if successful and non-zero if unsuccessful.

Errors:

If the remove operation was unsuccessful, `remove` returns a non-zero value.

Description:

`remove` deletes the file identified by the string pointer `filename`. If the file is open it will be deleted only if this is permitted by the host system.

`remove` is not included in the reduced library.

See also:

`rename`

rename Renames a file.

Synopsis:

```
#include <stdio.h>
int rename(const char *old, const char *new);
```

Arguments:

`const char *old`	A pointer to the old filename.
`const char *new`	A pointer to the new filename.

Results:

Returns zero if **rename** was successful and non-zero if it was not.

Errors:

If the rename was unsuccessful, **rename** returns a non-zero value.

Description:

rename changes the name of the file from **old** string to **new** string. If a file with the new name already exists the existing file will only be overwritten if this is permitted by the host operating system.

rename is not included in the reduced library.

See also:

remove

`rewind` Sets the read/write pointer to the start of a file stream.

Synopsis:

```
#include <stdio.h>
void rewind(FILE *stream);
```

Arguments:

`FILE *stream`	A pointer to a file stream.

Results:

No value is returned.

Errors:

None.

Description:

`rewind` sets the read/write position pointer of the file stream `stream` to the start of the file. The error indicators for the stream are cleared.

`rewind` is not included in the reduced library.

Example:

```
#include <stdio.h>

int main( void )
{
   FILE *stream;

   stream = fopen("data.dat","w+");

   if (stream == NULL)
      printf("Couldn't open data.dat for write.\n");
   else
   {
      fprintf(stream, "01234");
      rewind(stream);
      printf("First character in data.dat is:
             '%c'\n", getc(stream));
   }
}
```

```
/*
 * Output:
 *          First character in data.dat is '0'
 */
```

See also:

`fsetpos`

`scanf` Reads formatted data from standard input.

Synopsis:

```
#include <stdio.h>
int scanf(const char *format, ...);
```

Arguments:

`const char *format`	A format string.
`...`	Subsequent arguments to the format string.

Results:

Returns the number of inputs which have been successfully converted. If an end-of-file character occurred before any conversions took place, `scanf` returns EOF.

Errors:

If an end of file character occurred before any conversions took place, `scanf` returns EOF. Other failures cause termination of the procedure.

Description:

`scanf` matches the data read from the standard input to the specifications set out by the format string, `format`. The format string can include white space, ordinary characters, or conversion tokens:

1. White space causes the next series of white space characters read to be ignored.

2. Ordinary characters in the format string cause the characters read to be compared to the corresponding character in the format string. If the characters do not match, conversion is terminated.

3. A conversion token in the format string causes the data sequence read in to be checked to see if it is in the specified format. If it is, it is converted and placed in the appropriate argument following the format string. If the data is not in the correct format, conversion is terminated.

The meaning of the format string is as described for `fscanf`.

Any mismatch between the token format and the data received causes an early termination of `scanf`.

`scanf` is not included in the reduced library.

See also:

`fscanf`

segread Reads host processor segment registers. DOS only.

Synopsis:

```
#include <dos.h>
void segread(struct SREGS *segregs);
```

Arguments:

`struct SREGS *segregs`	The read-in values of the segment registers.

Results:

Returns no result.

Errors:

Any error sets `errno` to the value `EDOS`. Any attempt to use `segread` on operating systems other than DOS also sets `errno`. Failure of the function also generates the server error message:

```
[Encountered unknown primary tag (50)]
```

Description:

`segread` reads the current values of the host 80*x*86 processor's segment registers into `segregs`.

`segread` is not included in the reduced library.

See also:

`intdos` `intdosx`

SemAlloc Allocates and initialises a semaphore.

Synopsis:

```
#include <semaphor.h>
Semaphore *SemAlloc(int value);
```

Arguments:

`int value`	The initial value of the semaphore.

Results:

Returns a pointer to an initialised semaphore.

Errors:

If space cannot be allocated `SemAlloc` returns a NULL pointer.

Description:

Allocates space for a semaphore and returns a pointer to it. The semaphore is set to the `value` parameter.

See also:

`SemInit`

`SemInit` Initialises an existing semaphore.

Synopsis:

```
#include <semaphor.h>
void SemInit(Semaphore *sem, int value);
```

Arguments:

`Semaphore *sem`	A pointer to a semaphore.
`int value`	The initial value of the semaphore.

Results:

Returns no result.

Errors:

None.

Description:

SemInit initialises the semaphore pointed to by `sem` and assigns to it the initial value `value`.

See also:

`SemAlloc`

`SemSignal` Releases a semaphore.

Synopsis:

```
#include <semaphor.h>
void SemSignal(Semaphore *sem);
```

Arguments:

`Semaphore *sem`	A pointer to a semaphore.

Results:

Returns no result.

Errors:

None.

Description:

Releases the semaphore pointed to by `sem` and runs the next process on the semaphore's queue. If no processes are waiting on the queue the semaphore value is incremented.

See also:

`SemWait`

`SemWait` Acquires a semaphore.

Synopsis:

```
#include <semaphor.h>
void SemWait(Semaphore *sem);
```

Arguments:

`Semaphore *sem`	A pointer to a semaphore.

Results:

Returns no result.

Errors:

None.

Description:

Blocks the current process if the semaphore is already set to zero (acquired), otherwise acquires the semaphore, decrements it, and continues the process. Blocked processes do not continue until the semaphore is released by a call to `SemSignal` by another process.

See also:

`SemSignal`

server_transaction Calls any ISERVER function.

Synopsis:

```
#include <iocntrl.h>
int server_transaction(char *message, int length,
                       char *reply);
```

Arguments:

`char *message`	The server packet to be sent.
`int length`	The length of the server packet.
`char *reply`	A pointer to an array where the reply packet is to be stored.

Results:

Returns the length in bytes of the server reply packet, or −1 if an error occurs.

Errors:

Error codes returned are as follows:

1 Length is less than the minimum server transaction of 8 bytes.

2 Length is greater than 510.

3 Length is not an even number.

Description:

The runtime library provides functions which access a defined *sub*set of ISERVER functions. Some server functions are therefore not directly accessible by C function calls.

`server_transaction` allows controlled access to any ISERVER function from a C program. It allows the full functionality of the supplied ISERVER to be used from C and supports the calling of user-defined functions and alternative servers. A list of callable functions supplied with the standard toolset ISERVER can be found in appendix D '*ISERVER protocol*' of the accompanying User Manual.

`server_transaction` sends the packet pointed to by `message`, of length `length`, to the server. The server reply is stored in the array pointed to by `reply`.

For those familiar with occam, `server_transaction` performs the equivalent of the following occam output and input statements:

```
ToServer   ! length::message
FromServer ? replylen::reply
```

where: `ToServer` and `FromServer` are the server channels.

`length` and `replylength` are the packet lengths and `message` and `reply` are the data packets themselves.

`replylen` is the value returned by the function if no error occurs.

`server_transaction` provides low level access to the server in a secure manner. The user constructed packet is forwarded to the server, and the reply sent, via *protected* channels.

Note: There is no protection against the message and reply pointers being the same, in which case the original message packet is overwritten.

The following example uses `server_transaction` to obtain the transputer board size by calling the **Getenv** server function.

The structure of the packet to request the boardsize environment variable is given below. Numbers along the top row are Byte numbers.

0	1	2	3	4	5	6	7	8	9	10	11	12
32	10	00	I	B	O	A	R	D	S	I	Z	E

Byte 0 is the tag of the **Getenv** function. Bytes 1 and 2 make up a 16 bit number which represents the length of the string `IBOARDSIZE`. The string follows from byte 3 onwards.

The reply packet is similar except that byte 0 is the result byte and the string contains the value of the environment variable.

Example:

```
#include <misc.h>
#include <stdio.h>

int main()
{
  char message[512], reply[512];
  /* 512 byte buffers */
  char *name = "IBOARDSIZE";
```

```
        /* The env variable of interest */
        int length, i;

        /* set up packet to send */
        message[0] = 32;                /* getenv tag */
        message[1] = strlen(name);
        /* length of env variable name */
        message[2] = 0;
        strcpy(&message[3], name);
        /* calculate total length of packet */
        length = 3 + strlen(name);
        /* make sure length is an even number */
        length = (length + 1) & ~1;
        /* perform the transaction */
        length = server_transaction(message, length, reply);
        /* process reply */
        if (length == -1)
          printf("error in server transaction\n");
        else
        {
          /* print out result byte */
          printf("result = %d\n", reply[0]);
          /* print out length of env variable value */
          printf("length of result string = %d\n",reply[1]);
          /* terminate the result string */
          reply[(int)reply[1] + 3] = '\0';
          /* print out the result string */
          printf("string = [%s]\n", &reply[3]);
        }
      }
```

set_abort_action Sets/queries action taken by abort.

Synopsis:

```
#include <misc.h>
int set_abort_action(int mode);
```

Arguments:

int mode	The mode to be set.

Results:

Returns the previous termination mode (the mode in operation before set_abort_action was called).

Errors:

None.

Description:

Sets, or queries, the mode of termination for abort. mode can have any of the following values:

ABORT_EXIT	Causes a normal abort (without halting the transputer).
ABORT_HALT	Causes abort to halt the transputer.
ABORT_QUERY	Returns the current abort mode. Leaves the mode unchanged.

If ABORT_HALT is used abort first enables HALT mode by setting the Halt-On-Error flag and then sets the processor Error flag. When the transputer halts the following message is displayed by the server:

```
Error:  Transputer error flag has been set.
```

Note: Care should be taken when calling set_abort_action in a concurrent environment. Calls to the function by independently executing, unsynchronised processes may change the abort action. set_abort_action should normally be called at the start of the program to set the action of abort for the entire program.

See also:

abort

setbuf Controls file buffering.

Synopsis:

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
```

Arguments:

`FILE *stream`	A pointer to a file stream.
`char *buf`	A pointer to an array of size `BUFSIZ`.

Results:

Returns no value.

Errors:

None.

Description:

`setbuf` may be called after the file associated with `stream` has been opened, but before it has been read from or written to. `setbuf` causes `stream` to be fully buffered in the array `buf`. It is equivalent to a call to `setvbuf` with the values `_IOFBF` for `mode` and `BUFSIZ` for `size`. If `buf` is a NULL pointer, the stream will not be buffered.

`setbuf` is not included in the reduced library.

See also:

`setvbuf`

setjmp Sets up a non-local jump.

Synopsis:

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Arguments:

`jmp_buf env`	An array into which a copy of the calling environment is put.

Results:

When first called, `setjmp` stores the calling environment in `env` and returns zero. After a subsequent call to `longjmp` it returns a value set by `longjmp`, which is always non-zero.

Errors:

The `setjmp` function should only appear in one of the following contexts:

- The entire controlling expression of a selection or iteration statement.
- One operand of a relational or equality operator with the other operand being an integral constant expression. The resultant expression controls a selection or iteration statement.
- The operand of a unary ! operator. The resultant expression controls a selection or an iteration statement.
- The complete expression of an expression statement.

Description:

`setjmp` is used to set up a non-local goto by saving the calling environment in `env`. This environment is used by the `longjmp` function.

When first called, `setjmp` stores the calling environment in `env` and returns zero. A subsequent call to `longjmp` using `env` will cause execution to continue as if the call to `setjmp` had just returned with the value given in the call to `longjmp`. This value will always be non-zero.

See also:

`longjmp`

setlocale
Sets or interrogates part of the program's locale.

Synopsis:

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

Arguments:

`int category`	A specification of which part of the locale is to be set or interrogated.
`const char *locale`	A pointer to the string which selects the environment of the locale.

Results:

Returns "C" if `locale` is NULL, if `*locale` is NULL, or if `*locale` is "C". Otherwise returns NULL.

Errors:

Returns NULL if the parameters are invalid.

Description:

`setlocale` sets or interrogates part of the program's locale according to the values of `category` (the part to be set) and `locale` (a pointer to a string describing the environment to which it is to be set).

`category` can take the following values:

1	`LC-ALL`	All categories.
2	`LC_COLLATE`	Affects `strcoll` and `strxfrm`.
3	`LC_CTYPE`	Affects character handling
4	`LC_NUMERIC`	Affects the format of the decimal point (e.g., '.' ',', etc).
5	`LC_TIME`	Affects the `strftime` function.
6	`LC_MONETARY`	Affects monetary formatting information.

If `locale` is a null string, `setlocale` returns the current locale for the given category. In the current implementation the only acceptable locale is "C".

See also:

`localeconv`

setvbuf Defines the way that a file stream is buffered.

Synopsis:

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode,
            size_t size);
```

Arguments:

`FILE *stream`	A pointer to a file stream.
`char *buf`	A pointer to a file buffer.
`int mode`	The way the file stream is to be buffered.
`size_t size`	The size of the file buffer.

Results:

`setvbuf` returns zero if successful, and non-zero if the operation fails.

Errors:

If `mode` or `size` is invalid, or `stream` cannot be buffered, `setvbuf` returns a non-zero value.

Description:

`setvbuf` may be called after the file associated with `stream` has been opened, but before it has been read from or written to. `setvbuf` causes `stream` to be buffered in the format specified by `mode`. Valid formats are:

1	`_IOFBF`	Fully buffered I/O
2	`_IOLBF`	Line buffered output
3	`_IONBF`	Unbuffered I/O

The buffer used is of `size` bytes. If `buf` is not a NULL pointer, it is used as the buffer, otherwise an internally allocated array is used.

`setvbuf` is not included in the reduced library.

See also:

`setbuf`

`signal` Defines the way that errors and exceptions are handled.

Synopsis:

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

Arguments:

`int sig`	A signal number (a predefined value, describing an error/exception type).
`void (*func)(int)`	A function which is invoked on reception of `sig`.

Results:

If the signal number is recognised a pointer to the function previously associated with the signal number `sig` is returned, otherwise `SIG_ERR` is returned.

Errors:

If the predefined error/exception value is not recognised by `signal`, `signal` returns `SIG_ERR` and sets `errno` to the value `ESIGNUM`.

Description:

`signal` specifies the functions to be called on reception of particular, predetermined signal values.

`func` can be any user-defined function, or one of the following two predefined functions which are implemented as macros in the `signal.h` header file:

`SIG_DFL`	Uses the default system error/exception handling for the predefined value.
`SIG_IGN`	Ignores the error/exception.

The functions will then be called in response to a "`raise`" or other invocation of the signal handler, using a signal number as a parameter. If the second parameter is a function other than `SIG_DFL` or `SIG_IGN`, `SIG_DFL` will be called, and then the function.

When a signal is raised the default signal handling is reset by a call of the form `signal(sig, SIG_DFL)` and then the signal handler function is called. If `sig` takes the value `SIGILL` then the default resetting still occurs.

The available signal numbers are as follows:

1	`SIGABRT`	Abort error
2	`SIGFPE`	Arithmetic exception
3	`SIGILL`	Illegal instruction
4	`SIGINT`	Attention request from user
5	`SIGSEGV`	Bad memory access
6	`SIGSTERM`	Termination request
8	`SIGIO`	Input/output possible
9	`SIGURG`	Urgent condition on I/O channel
10	`SIGPIPE`	Write on pipe with no corresponding read
11	`SIGSYS`	Bad argument to system call
12	`SIGALRM`	Alarm clock
13	`SIGWINCH`	Window changed
14	`SIGLOST`	Resource lost
15	`SIGUSR1`	User defined signal
16	`SIGUSR2`	User defined signal
17	`SIGUSR3`	User defined signal

The default handling and handling at program startup for all signals except `SIGABRT` and `SIGTERM` is no action. For `SIGABRT` the handling depends on `set_abort_action`, and for `SIGTERM` the program is terminated via a call to `exit` with the parameter `EXIT_FAILURE`.

Example:

```
/*
 * To arrange that an interrupt by the user
 * should not go through the default exception
 * handling system, call
 *
 *      signal( SIGILL, SIG_IGN )
 *
 * If the signal is then raised in a
 * later part of the program:
 *
 *      raise( SIGILL )
 *
 * the signal will be ignored.
 */
```

Note: Care should be taken when using `signal` in a concurrent environment. Although simultaneous access to the function is controlled through a semaphore,

the registration of a function with the *same* signal number, for example by independent parallel processes overrides the previous value.

See also:

`raise`

`sin` Calculates the sine of the argument.

Synopsis:

```
#include <math.h>
double sin(double x);
```

Arguments:

`double x` A number in radians.

Results:

Returns the sine of `x` in radians.

Errors:

None.

Description:

`sin` calculates the sine of a number (given in radians).

`sinf` Calculates the sine of a `float` number.

Synopsis:

```
#include <mathf.h>
float sinf(float x);
```

Arguments:

`float x` A number in radians.

Results: Returns the sine of `x` in radians.

Errors:

None.

Description: `float` form of `sin`.

See also:

`sin`

sinh Calculates the hyperbolic sine of the argument.

Synopsis:

```
#include <math.h>
double sinh(double x);
```

Arguments:

`double x`	A number.

Results:

Returns the hyperbolic sine of `x`.

Errors:

A range error will occur if `x` is so large that `sinh` would result in an overflow. In this case `sinh` returns the value `HUGE_VAL` (with the same sign as the correct value of the function) and `errno` is set to `ERANGE`.

Description:

`sinh` calculates the hyperbolic sine of a number.

`sinhf` Calculates the hyperbolic sine of a `float` number.

Synopsis:

```
#include <mathf.h>
float sinhf(float x);
```

Arguments:

`float x` A number.

Results: Returns the hyperbolic sine of `x`.

Errors: A range error will occur if `x` is so large that `sinhf` would result in an overflow. In this case `sinhf` returns the value `HUGE_VAL_F` (with the same sign as the correct value of the function) and `errno` is set to `ERANGE`.

Description: `float` form of `sinh`.

See also:

`sinh`

sprintf

Writes a formatted string to a string.

Synopsis:

```
#include <stdio.h>
int sprintf(char *s, const char *format, ...);
```

Arguments:

`char *s`	A string that the output is written to.
`const char *format`	A format string.
`...`	Subsequent arguments to the format string.

Results:

Returns the number of characters written, excluding the string terminating character.

Errors:

None.

Description:

`sprintf` writes the string pointed to by `format` to `s`. When `sprintf` encounters a percent sign (%) in the format string, it expands the equivalent argument into the format defined by the tokens after the %.

For the meaning of the format string see the description of `fprintf`.

Each token acts on the equivalent argument, that is, the third token relates to the third argument after the format string. There must be a single argument for each token. If the token or its equivalent argument is invalid, the behaviour is undefined.

To use `sprintf` in the reduced library include the header file `stdiored.h`.

See also:

`fprintf`

`sqrt` Calculates the square root of the argument.

Synopsis:

```
#include <math.h>
double sqrt(double x);
```

Arguments:

`double x` A number.

Results:

Returns the non-negative square root of `x`.

Errors:

A domain error will occur if `x` is negative. In this case `errno` is set to `EDOM`.

Description:

`sqrt` calculates the square root of a number.

`sqrtf` `float` form of `sqrtf`.

Synopsis:

```
#include <mathf.h>
float sqrtf(float x);
```

Arguments:

`float x`	A number.

Results:

Returns the non-negative square root of `x`.

Errors:

A domain error will occur if `x` is negative. In this case `errno` is set to `EDOM`.

Description:

`float` form of `sqrt`.

See also:

`sqrt`

`srand` Sets the seed for pseudo-random numbers generated by `rand`.

Synopsis:

```
#include <stdlib.h>
void srand(unsigned int seed);
```

Arguments:

`unsigned int seed` The new seed to be used by `rand`.

Results:

No value is returned.

Errors:

None.

Description:

`srand` causes `rand` to be seeded with the value seed. Subsequent calls to `rand` will start a new sequence of pseudo-random numbers. If `srand` is called again with the same value of `seed` the random number sequence will be repeated.

If `rand` is called before any calls to `srand` have been made the effect will be the same as if `srand` had been called with a `seed` value of 1.

See also:

`rand`

sscanf Reads formatted data from a string.

Synopsis:

```
#include <stdio.h>
int sscanf(const char *s, const char *format, ...);
```

Arguments:

`const char *s`	The string the data is read from.
`const char *format`	A format string.
`...`	Subsequent arguments to the format string.

Results:

Returns the number of inputs which have been successfully converted. If a string terminating character occurred before any conversions took place, `sscanf` returns `EOF`.

Errors:

If a string terminating character occurred before any conversions took place, `sscanf` returns `EOF`. Other failures cause termination of the procedure.

Description:

`sscanf` matches the data read from the string `s` to the specifications set out by the format string. The format string can include white space, ordinary characters, or conversion tokens, which are interpreted as follows:

- White space causes the next series of white space characters read to be ignored.
- Ordinary characters in the format string cause the characters read to be compared to the corresponding character in the format string. If the characters do not match, conversion is terminated.
- A conversion token in the format string causes the data sequence read in to be checked to see if it is in the specified format. If it is, it is converted and placed in the appropriate argument following the format string. If the data is not in the correct format, conversion is terminated.

The conversion tokens are those described in `fscanf`.

Each token acts on the equivalent argument, that is, the third token relates to

the third argument after the format string. There must be a single conversion sequence received for each token. If the token is invalid, the behaviour is undefined.

Any mismatch between the token format and the data received causes an early termination of `sscanf`.

To use `sscanf` in the reduced library include the header file `stdiored.h`.

See also:

`fscanf`

`strcat` Appends one string to another.

Synopsis:

```
#include <string.h>
char *strcat(char *s1, const char *s2);
```

Arguments:

`char *s1`	A pointer to the string to be extended.
`const char *s2`	A pointer to the string to be appended.

Results:

Returns the unchanged value of `s1`.

Errors:

None.

Description:

`strcat` appends the string pointed to by `s2` (including the null terminating character) onto the end of the string pointed to by `s1`. The first character of `s2` overwrites the null terminating character of `s1`.

See also:

`strncat`

`strchr` Finds the first occurrence of a character in a string.

Synopsis:

```
#include <string.h>
char *strchr(const char *s, int c);
```

Arguments:

`const char *s`	A pointer to the string to be searched.
`int c`	The character to be searched for.

Results:

If the character is found, `strchr` returns a pointer to the matched character. It returns a null pointer if the character `c` is not in the string.

Errors:

None.

Description:

`strchr` finds the first occurrence of `c` in the string pointed to by `s`. The search includes the null terminating character. `c` is converted to a `char` before the search begins.

Example:

```
char string[50] = "fdakjrejnij";
char *n_pointer;

n_pointer = strchr(string, 'n');
```

See also:

`memchr strpbrk strrchr`

`strcmp` Compares two strings.

Synopsis:

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

Arguments:

`const char *s1`	A pointer to one of the strings to be compared.
`const char *s2`	A pointer to the other string to be compared.

Results:

Returns the following :

> A negative integer if the `s1` string is numerically less than the `s2` string.
>
> A zero value if the two strings are numerically the same.
>
> A positive integer if the `s1` string is numerically greater than the `s2` string.

Errors:

None.

Description:

`strcmp` compares the two strings pointed to by `s1` and `s2`. The comparison is of the numerical values of the ASCII characters.

See also:

`memcmp strcoll strncmp`

strcoll

Compares two strings (transformed according to the program's locale).

Synopsis:

```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

Arguments:

`const char *s1`	A pointer to one of the strings to be compared.
`const char *s2`	A pointer to the other string to be compared.

Results:

Returns the following:

A negative integer if the `s1` string is numerically less than the `s2` string.

A zero value if the two strings are numerically the same.

A positive integer if the `s1` string is numerically greater than the `s2` string.

Errors:

None.

Description:

`strcoll` compares the two strings pointed to by `s1` and `s2`. Before comparison takes place the two strings are transformed according to the `LC_COLLATE` category of the program's locale. Since the only permissible locale in the current implementation is "C", `strcoll` is equivalent to `strcmp`.

The string comparison is of the characters' numerical ASCII codes.

See also:

`memcmp` `strcmp` `strncmp`

`strcpy` Copies a string into an array.

Synopsis:

```
#include <string.h>
char *strcpy(char *s1, const char *s2);
```

Arguments:

`char *s1`	A pointer to the array used as the copy destination.
`const char *s2`	A pointer to the string used as the copy source.

Results:

Returns the unchanged value of `s1`.

Errors:

The behaviour of `strcpy` is undefined if the source and destination overlap.

Description:

`strcpy` copies the source string (pointed to by `s2`) into the destination array (pointed to by `s1`). The copy includes the null terminating character. The behaviour of `strcpy` is undefined if the source and destination overlap.

Calls to `strcpy` can be replaced by the compiler predefine `_strcpy` by redefining the function name. `_strcpy` is implemented directly as transputer assembly code in selected cases. For further details see section 11.4 in the accompanying User Manual.

See also:

`strncpy` `_strcpy`

strcspn

Counts the number of characters at the start of a string which do not match any of the characters in another string.

Synopsis:

```
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
```

Arguments:

`const char *s1`	A pointer to the string to be measured.
`const char *s2`	A pointer to the string containing the characters to be checked.

Results:

Returns the length of the unmatched segment.

Errors:

None.

Description:

`strcspn` counts the characters in the string pointed to by `s1` which are not in the string pointed to by `s2`. As soon as `strcspn` finds a character present in both strings it stops and returns the number of characters counted.

The null terminating character is not considered to be part of the `s2` string.

Example:

```
#include <stdio.h>
#include <string.h>

int main( void )    /* Print string up to any
                       numeric characters. */
{
   char *dec_string = "1234567890";
   char *given_string = "Hello there 123hello";
   size_t no_chars;
   no_chars = strcspn(given_string, dec_string);
   given_string[no_chars] = '\0';
   puts(given_string); /* prints "Hello there" */
```

```
    }
```

See also:

`strspn strtok`

strerror Converts an error number into an error message string.

Synopsis:

```
#include <string.h>
char *strerror(int errnum);
```

Arguments:

`int errnum` The error number to be converted.

Results:

Returns a pointer to the error message string.

Errors:

None.

Description:

`strerror` generates one of the following error messages according to the value of `errnum`:

Value of `errnum`	Message
`EDOM`	**EDOM - function argument out of range**
`ERANGE`	**ERANGE - function result not representable**
`ESIGNUM`	**ESIGNUM - illegal signal number to signal()**
`EIO`	**EIO - error in low level server I/O**
`EFILPOS`	**EFILPOS - error in file positioning functions**
0	**No error (errno = 0)**

If `errnum` is not one of the above values the following error is generated:

Error code (errno) *errnum* **has no associated message**

Note: Care should be taken when calling `strerror` in a concurrent environment. Calls to the function by independently executing, unsynchronised processes may corrupt the returned error string.

See also:

`perror`

strftime

Does a formatted conversion of a **tm** structure to a string.

Synopsis:

```
#include <time.h>
size_t strftime(char *s, size_t maxsize,
                const char *format,
                const struct tm *timeptr);
```

Arguments:

`char *s`	A pointer to the buffer where the string is written.
`size_t maxsize`	The maximum number of characters to be written into the string.
`const char *format`	A pointer to the format string.
`const struct tm *timeptr`	A pointer to a calendar time structure.

Results:

If the number of characters written is less than **maxsize**, **strftime** returns the number of characters written, otherwise **strftime** returns zero (0).

Errors:

If the number of characters to be written exceeds **maxsize**, **strftime** returns zero, and the contents of the string pointed to by **s** are undefined.

Description:

strftime is used to convert the values in a time structure according to the demands of a format string, and to write the resulting string to a string. The format string consists of ordinary characters and tokens. Normal characters are written directly to **s**, and tokens are expanded. Tokens are single characters, preceded by the percent character **%**.

Token	Meaning	Range
%a	Abbreviated day	(Mon – Sun).
%A	Full day	(Monday – Sunday).
%b	Abbreviated month	(Jan – Dec).
%B	Full month	(January – december).
%c	Date and time in form of a string of decimal numbers.	(e.g. Sun Jul 23 11:27:32 1989).
%d	Day of the month as a decimal number.	01 – 31
%H	Hours using twenty-four hour clock.	00 – 23
%	Hours using twelve hour clock.	01 – 12
%j	Day of the year.	001 – 366
%m	Month as a decimal number.	01 – 12
%M	Minutes.	00 – 59
%p	AM or PM.	
%S	Seconds.	00 – 61
%U	Week number, counting Sunday as first day of week one.	00 – 53.
%w	Day of week, counting from Sunday.	0 – 6
%W	Week number, counting Monday as first day	00 – 53.
%x	Date in default format.	(e.g. Sun Jul 23 1989).
%X	Time in default format.	(e.g. 11:27:32).
%y	Year without century.	00 – 99
%Y	Year with century.	e.g. 1989
%Z	Time zone if one exists.	–
%%	'%'.	–

Example:

```
#include <stdio.h>
#include <time.h>

/* Display the day in different ways */

int main( void )
{
   char day_line[300];
   struct tm *calendar;
   time_t current;

   time( &current );
   calendar = localtime( &current );
   strftime (day_line, 300,
             "Different days are %a, %A, %j, %d, %w",
             calendar);
   printf(day_line);
}
```

See also:

asctime ctime localtime clock difftime mktime time

strlen Calculates the length of a string.

Synopsis:

```
#include <string.h>
size_t strlen(const char *s);
```

Arguments:

`const char *s` A pointer to the string to be measured.

Results:

Returns the length of the string (excluding the NULL terminating character).

Errors:

None.

Description:

`strlen` counts the number of characters in the string up to, but not including, the NULL terminating character.

Example:

```
char *string = "String to be measured";
size_t result;

result = strlen(string);


/*
  Gives a result of 21
*/
```

strncat

Appends one string onto another (up to a maximum number of characters).

Synopsis:

```
#include <string.h>
char *strncat(char *s1, const char *s2,
              size_t n);
```

Arguments:

`char *s1`	A pointer to the string to be extended.
`const char *s2`	A pointer to the string to be appended.
`size_t n`	The maximum number of characters to be appended.

Results:

Returns the unchanged value of `s1`.

Errors:

None.

Description:

`strncat` copies a maximum of `n` characters from the string pointed to by `s2` (excluding the null terminating character) onto the end of the string pointed to by `s1`. The first character of `s2` overwrites the null terminating character of `s1`. A null terminating character is appended to the end of the result.

See also:

`strcat`

strncmp Compares the first n characters of two strings.

Synopsis:

```
#include <string.h>

int strncmp(const char *s1, const char *s2,
            size_t n);
```

Arguments:

`const char *s1`	A pointer to one of the strings to be compared.
`const char *s2`	A pointer to the other string to be compared.
`size_t n`	The maximum number of characters to be compared.

Results:

Returns:

A negative integer if the `s1` string is numerically less than the `s2` string.

A zero value if the two strings are numerically the same.

A positive integer if the `s1` string is numerically greater than the `s2` string.

Errors:

None.

Description:

`strncmp` compares up to the first `n` characters of the strings pointed to by `s1` and `s2`.

The comparison is of the numerical values of the ASCII characters.

Example:

```
/*
  Compares two strings
*/

char string1[50], string2[50];
int result;
```

```
strcpy(string1, "Text");
strcpy(string2, "Textual difference");
result = strncmp(string1, string2, 4);

/*
strncmp returns 0
*/
```

See also:

memcmp strcmp strcoll strncmp

`strncpy`

Copies a string into an array (to a maximum number of characters).

Synopsis:

```
#include <string.h>
char *strncpy(char *s1, const char *s2, size_t n);
```

Arguments:

`char *s1`	A pointer to the array used as the copy destination.
`const char *s2`	A pointer to the string used as the copy source.
`size_t n`	The maximum number of characters to be copied.

Results:

Returns the unchanged value of `s1`.

Errors:

The behaviour of `strncpy` is undefined if the source and destination overlap.

Description:

`strncpy` copies up to `n` characters from the source string (pointed to by `s2`) into the destination array (pointed to by `s1`). The behaviour of `strcpy` is undefined if the source and destination overlap.

If the source string is less than `n` characters long, the extra spaces in the destination array will be filled with null characters.

See also:

`strcpy`

strpbrk

Finds the first character in one string present in another string.

Synopsis:

```
#include <string.h>
char *strpbrk(const char *s1, const char *s2);
```

Arguments:

`const char *s1`	A pointer to the string to be searched.
`const char *s2`	A pointer to the string containing the characters to be searched for.

Results:

Returns a pointer to the first character found in both strings. If none of the characters in the `s2` string occur in the `s1` string, `strpbrk` returns a null pointer.

Errors:

None.

Description:

`strpbrk` finds the first character in the string pointed to by `s1` which is also contained within the string pointed to by `s2`.

Example:

```
/* Return a pointer to the first occurrence of
                            'r', 'c', or 'm', */

#include <stdio.h>
#include <string.h>

int main()
{
  char *string = "The Inmos C Compiler";
  char *result;

  result = strpbrk(string, "rcm");
  printf("%s\n", result);
}

/* result = "mos C Compiler" */
```

See also:

`strchr` `strrchr`

strrchr

Finds the last occurrence of a given character in a string.

Synopsis:

```
#include <string.h>
char *strrchr(const char *s, int c);
```

Arguments:

`const char *s`	A pointer to the string to be searched.
`int c`	The character to be searched for.

Results:

Returns a pointer to the last occurrence of the character.

Errors:

Returns NULL if `c` does not occur in the string.

Description:

`strchr` finds the last occurrence of `c` in the string pointed to by `s`. The search includes the null terminating character. `c` is converted to a char before the search begins.

Example:

```
/*  Finds the last time that '9' occurs
                         in a string */

#include <stdio.h>
#include <string.h>

int main()
{
   char *string = "9 times 9 = 81";
   char *result;

   result = strrchr(string, '9');
   printf("%s\n", result );
                /* result = "9 = 81" */
}
```

See also: `strpbrk` `strchr`

strspn

Counts the number of characters at the start of a string which are also in another string.

Synopsis:

```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

Arguments:

`const char *s1`	A pointer to the string to be measured.
`const char *s2`	A pointer to the string containing the characters to be looked for.

Results:

Returns the length of the matched segment.

Errors:

None.

Description:

`strspn` counts the characters in the string pointed to by `s1` which are also present in the string pointed to by `s2`. As soon as `strspn` finds a character in the first string which is not present in the second string, it stops and returns the number of characters counted.

Example:

```
#include <string.h>
#include <stdio.h>

int main( void )
{
   char *string = "cracking";
   size_t result;

   result = strspn(string, "arc");
   printf("%d\n", result ); /* 4 in this case */
}
```

See also:

`strcspn strtok`

strstr Finds the first occurrence of one string in another.

Synopsis:

```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

Arguments:

`const char *s1`	A pointer to the string to be searched.
`const char *s2`	A pointer to the string to be searched for.

Results:

Returns a pointer to the string, if found. If `s2` points to a string of zero length, the function returns `s1`. If the `s2` string does not occur within the `s1` string the function returns NULL.

Errors:

None.

Description:

`strstr` finds the first occurrence of the `s2` string (excluding the null terminating character) in the `s1` string.

Example:

```
#include <string.h>
#include <stdio.h>

int main()
{
   char *string1 = "string to be searched";
   char *string2 = "sea";

   printf("%s\n", strstr(string1, string2));
 }

/* Displays "searched" */
```

See also:

`strpbrk` `strspn`

strtod

Converts the initial part of a string to a double and saves a pointer to the rest of the string.

Synopsis:

```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr);
```

Arguments:

`const char *nptr`	A pointer to the string to be converted.
`char **endptr`	A pointer to the location which is to receive a pointer to the rest of the string.

Results:

Returns the converted value if the conversion is successful. If no conversion is possible, `strtod` returns zero.

Errors:

If the result would cause overflow, `errno` is set to `ERANGE` and the value `HUGE_VAL` is returned. If the result would cause underflow, `errno` is set to `ERANGE` and zero is returned.

Description:

`strtod` converts the initial part of the string pointed to by `nptr` to a number represented as a double. `strtod` expects the string to consist of the following sequence:

1. Leading white space (optional).
2. A plus or minus sign (optional).
3. A sequence of decimal digits, which may contain a decimal point.
4. An exponent (optional) consisting of an 'E' or 'e' followed by an optional sign and a string of decimal digits.
5. One or more unrecognised characters (including the null string terminating character).

`strtod` ignores the leading white space, and converts all the recognised characters. If there is no decimal point or exponent part in the string, a decimal point is assumed after the last digit in the string.

The string is invalid if the first non-space character in the string is not one of the

following characters:

+ – . 0 1 2 3 4 5 6 7 8 9

If `endptr` is not NULL, and the conversion took place, a pointer to the unrecognised part of the string is stored in the location pointed to by `endptr`. If conversion did not take place, the location is set to the value of `nptr`.

Example:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
 char *array = "97824.3E+4Goodbye";
 char *number_end;
 double x;

 x = strtod(array, &number_end);
 printf("strtod gives %f\n", x);
 printf("Number ended at %s\n", number_end);
}

/*
Prints:
  strtod gives 978243000.000000
         Number ended at Goodbye
*/
```

See also:

`atof atoi atol strtol`

`strtok` Converts a delimited string into a series of string tokens.

Synopsis:

```
#include <string.h>
char *strtok(char *s1, const char *s2);
```

Arguments:

`char *s1`	A pointer to the string to be broken up.
`const char *s2`	A pointer to the delimiter string.

Results:

Returns a pointer to the first character of a token. A NULL pointer is returned if no token is found.

Errors:

None.

Description:

`strtok` is used to break up the string pointed to by `s1` into separate strings. The input string is assumed to consist of a series of tokens separated from one another by one of the characters in the delimiter string pointed to by `s2`.

When `strtok` is first called, each character in the string pointed to by `s1` is checked to see if it is also present in the delimiting string pointed to by `s2`. `strtok` recognises the first character which is not in the delimiter string as the start of the first token. If no such character is found it is assumed that there are no tokens in `s1`, and `strtok` returns a NULL pointer.

Having found the start of a token, the `strtok` function searches for the end of the token, represented by a character present in the delimiting string. If such a character is found, it is overwritten with the NULL terminating character and `strtok` saves a pointer to the following character for use in a subsequent call. If no such character is found the token extends to the end of the string. `strtok` returns a pointer to the first character of the token.

The next token from the string is extracted by calling `strtok` with a NULL pointer as the first parameter. This causes `strtok` to use the pointer saved during the previous execution.

Note: Care should be taken when calling `strtok` in a concurrent environment.

Calls to the function by independently executing, unsynchronised processes may change the returned token pointer.

Example:

```
#include <stdio.h>
#include <string.h>

int main()
{
   char *string = "String^of things,to,,be^split";
   char *token;

   token = strtok(string, "^ ,");
   while (token != NULL)
   {
      printf("Token found = %s\n", token);
      token = strtok(NULL, "^ ,");
   }
}
   /*
    *  Gives the output:
    *      Token found = String
    *      Token found = of
    *      Token found = things
    *      Token found = to
    *      Token found = be
    *      Token found = split
    */
```

strtol

Converts the initial part of a string to a long integer and saves a pointer to the rest of the string.

Synopsis:

```
#include <stdlib.h>
long int strtol(const char *nptr,
                char **endptr, int base);
```

Arguments:

`const char *nptr`	A pointer to the string to be converted.
`char **endptr`	A pointer to the location which is to receive a pointer to the rest of the string.
`int base`	The radix representation of the integer string to be converted.

Results: Returns the converted value if the conversion is successful. If no conversion is possible, `strtol` returns zero. If the result would cause overflow the value `LONG_MAX` or `LONG_MIN` is returned (depending on the sign of the result).

Errors: If the result would cause overflow the value `LONG_MAX` or `LONG_MIN` is returned (depending on the sign of the result), and `errno` is set to `ERANGE`.

Description: `strtol` converts the initial part of the string pointed to by `nptr` to a long integer. `strtol` expects the string to consist of the following:

1. Leading white space (optional).
2. A plus or minus sign (optional).
3. An octal '0' or hexadecimal '0x' or '0X' prefix (optional).
4. A sequence of digits within the range of the appropriate base. The letters 'a' to 'z', and 'A' to 'Z' may be used to represent the values 10 to 35. For example, if base is set to 18, the characters for the values 0 to 17 ('0' to '9' and 'a' to 'h' or 'A' to 'H') are permitted.
5. One or more unrecognised characters (including the null string terminating character).

`strtol` ignores leading blanks, and converts all recognised characters.

The string is invalid if the first non-space character in the string is not a sign, an octal or hexadecimal prefix, or one of the permitted characters.

If `endptr` is not NULL, and the conversion took place, a pointer to the rest of

the string is stored in the location pointed to by `endptr`. If no conversion was possible, and `endptr` is not NULL, the value of `nptr` is stored in that location.

Example:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   char *array = "12345abcGoodbye";
   char *number_end;
   int base;
   long l;

   for(base = 2; base < 12; base += 3)
   {
      l = strtol(array, &number_end, base);
      printf("base = %d, strtol gives %ld\n",
              base, l);
      printf("Number ended at %s\n\n", number_end);
   }
}


/* Prints  base = 2, strtol gives 1
 *         Number ended at 2345abcGoodbye


 *         base = 5, strtol gives 194
 *         Number ended at 5abcGoodbye


 *         base = 8, strtol gives 5349
 *         Number ended at abcGoodbye

 *         base = 11, strtol gives 194875
 *         Number ended at bcGoodbye
 */
```

See also:

`atoi atol strtod strtoul`

strtoul

Converts the initial part of a string to an unsigned long int and saves a pointer to the rest of the string.

Synopsis:

```
#include <stdlib.h>
unsigned long int strtoul(const char *nptr,
                          char **endptr, int base);
```

Arguments:

`const char *nptr`	A pointer to the string to be converted.
`char **endptr`	A pointer to the location which is to receive a pointer to the rest of the string.
`int base`	The radix representation of the integer string to be converted.

Results:

Returns the converted value if the conversion is successful. If no conversion is possible, `strtoul` returns zero. If the result would cause overflow the value `ULONG_MAX` is returned.

Errors:

If the result would cause overflow the value `ULONG_MAX` is returned and errno is set to `ERANGE`.

Description:

`strtoul` converts the initial part of the string pointed to by `nptr` to an unsigned long int. `strtoul` expects the string to consist of the following:

1. Leading white space (optional).
2. An octal '0' or hexadecimal '0x' or '0X' prefix (optional).
3. A sequence of digits within the range of the appropriate base. The letters 'a' to 'z', and 'A' to 'Z' may be used to represent the values 10 to 35. For example, if base is set to 18, the characters for the values 0 to 17 ('0' to '9' and 'a' to 'h' or 'A' to 'H') are permitted.
4. One or more unrecognised characters (including the NULL string terminating character).

`strtoul` ignores the leading white space, and converts all the recognised characters.

The string is invalid if the first non-space character in the string is not an octal or hexadecimal prefix, or one of the permitted characters (signs are not permitted). If `endptr` is not NULL, and the conversion took place, a pointer to the rest of the string is stored in the location pointed to by `endptr`. If no conversion was possible, and `endptr` is not NULL, the value of `nptr` is stored in that location.

See also:

`atoi atol strtod strtol`

strxfrm

Transforms a string according to the locale and copies it into an array (up to a maximum number of characters).

Synopsis:

```
#include <string.h>
size_t strxfrm(char *s1, const char *s2, size_t n);
```

Arguments:

`char *s1`	A pointer to the array used as the copy destination.
`const char *s2`	A pointer to the string used as the copy source.
`size_t n`	The maximum number of characters to be copied.

Results:

If the string to be copied fits into the destination string, `strxfrm` returns the number of characters copied (excluding the NULL terminating character); otherwise it returns 0.

Errors:

None.

Description:

`strxfrm` copies up to `n` characters from the source string (pointed to by `s2`) into the destination array (pointed to by `s1`), after transforming the source string according to the program's locale. Since the only permissible locale is "C", `strxfrm` is equivalent to `strncpy`. The behaviour of `strxfrm` is undefined if the source and destination overlap.

If the source string is less than `n` characters long, the extra spaces in the destination array will be filled with NULL characters.

See also:

`strncpy`

system Passes a command to host operating system for execution.

Synopsis:

```
#include <stdlib.h>
int system(const char *string);
```

Arguments:

`const char *string` A pointer to the string to be passed to the host.

Results:

Returns a non-zero value if `string` is a NULL pointer (to indicate that there is a command processor). If `string` is not a NULL pointer `system` returns the return value of the command which is host-defined.

Errors:

None.

Description:

`system` passes the string pointed to by `string` to the host environment to be executed by a command processor. `string` can be any command defi command defined on the host system, but should not be a command which causes the transputer to be rebooted as this would overwrite the program execu executing the call.

If `string` is a NULL pointer the call to `system` is an enquiry as to whether there is a command processor.

`system` is not included in the reduced library.

Note: Issuing a command that boots a program onto the transputer running the current program causes the program to fail by overwriting the memory.

The mode of execution of the command is defined by the host system.

`tan` Calculates the tangent of the argument.

Synopsis:

```
#include <math.h>
double tan(double x);
```

Arguments:

`double x` A number in radians.

Results: Returns the tangent of `x` in radians.

Errors:

None.

Description: `tan` calculates the tangent of a number (given in radians).

See also:

`stan`

`tanf` Calculates the tangent of a `float` number.

Synopsis:

```
#include <mathf.h>
float tanf(float x);
```

Arguments:

`float x` A number in radians.

Results:

Returns the tangent of `x`.

Errors:

None.

Description:

`float` form of `tan`.

See also:

`tan`

`tanh` Calculates the hyperbolic tangent of the argument.

Synopsis:

```
#include <math.h>
double tanh(double x);
```

Arguments:

`double x` A number.

Results:

Returns the hyperbolic tangent of `x`.

Errors:

None.

Description:

`tanh` calculates the hyperbolic tangent of a number.

See also:

`tanhf`

`tanhf` Calculates the hyperbolic tangent of a `float` number.

Synopsis:

```
#include <mathf.h>
float tanhf(float x);
```

Arguments:

`float x` A number.

Results:

Returns the hyperbolic tangent of `x`.

Errors:

None.

Description:

`float` form of `tanh`.

See also:

`tanh`.

`time` Reads the current time.

Synopsis:

```
#include <time.h>
time_t time(time_t *timer);
```

Arguments:

`time_t *timer`	A pointer to a location where the current time can be stored.

Results:

Returns the value of the current time. If the current time is not available, `time` returns −1, cast to `time_t`.

Errors:

`time` returns (`time_t`)−1, if the current time is not available.

Description:

`time` returns the closest possible approximation to the current time, and loads it into the location pointed to by `timer`, unless `timer` is NULL.

`time` is not included in the reduced library.

See also:

`asctime ctime localtime strftime clock difftime mktime`

`tmpfile` Creates a temporary binary file.

Synopsis:

```
#include <stdio.h>
FILE *tmpfile(void);
```

Arguments:

None.

Results:

Returns a pointer to the newly created file stream, or a NULL pointer if the file could not be created.

Errors:

Returns a NULL pointer if the file cannot be created.

Description:

`tmpfile` attempts to create a temporary binary file in the *current* directory. If the file is successfully created it is opened for update, that is, in mode "wb+". The file will automatically be removed when the program terminates or the temporary file is explicitly closed.

`tmpfile` is not included in the reduced library.

See also:

`tmpnam`

`tmpnam` Creates a unique filename.

Synopsis:

```
#include <stdio.h>
char *tmpnam(char *s);
```

Arguments:

`char *s` A pointer to the destination string for the filename.

Results:

If `s` is a null pointer, `tmpnam` returns a pointer to an internal object containing the new filename. Otherwise the new filename is put in the string pointed to by `s`, and `tmpnam` returns the unchanged value `s`. In this case `s` must point to an array of at least `L_tmpnam` characters.

Errors:

The effect of calling `tmpnam` more than `TMP_MAX` times is undefined.

Description:

`tmpnam` creates a unique filename (that is, one which does not match any existing filename) in the *current* directory. A different string is created each time `tmpnam` is called. `tmpnam` may be called up to `TMP_MAX` times.

Note: Care should be taken when calling `tmpnam` in a concurrent environment. Calls to the function by independently executing, unsynchronised processes may corrupt the returned file pointer.

`tmpnam` is not included in the reduced library.

See also:

`tmpfile`

to86 Transfers transputer memory to the host. DOS only.

Synopsis:

```
#include <dos.h>
int to86(int len, char *here, pcpointer there);
```

Arguments:

`int len`	The number of bytes of transputer memory to be transferred.
`char *here`	A pointer to the transputer memory block.
`pcpointer there`	A pointer to the host memory block.

Results:

Returns the actual number of bytes transferred.

Errors:

Returns the number of bytes transferred until the error occurred and sets `errno` to the value `EDOS`. Any attempt to use `to86` on operating systems other than DOS also sets `errno` to `EDOS`. Failure of the function also generates the following server error message:

```
[Encountered unknown primary tag (50)]
```

Description:

`to86` transfers `len` bytes of transputer memory starting at `here` to a corresponding block starting at `there` in host memory. The function returns the number of bytes actually transferred. The host memory block used will normally have been previously allocated by a call to `alloc86`.

`to86` is not included in the reduced library.

See also:

`from86 alloc86`

`tolower` Converts upper-case letter to its lower-case equivalent.

Synopsis:

```
#include <ctype.h>
int tolower(int c);
```

Arguments:

`int c` The character to be converted.

Results:

Returns the lower-case equivalent of the given character. If the given character is not an upper-case letter it is returned unchanged.

Errors:

None.

Description:

`tolower` converts the character `c` to its lower-case equivalent. If `c` is not an upper-case letter it is not converted. Valid upper-case letters are ASCII characters in the range 'A' to 'Z'.

See also:

`toupper`

`toupper` Converts lower-case letter to its upper-case equivalent.

Synopsis:

```
#include <ctype.h>
int toupper(int c);
```

Arguments:

`int c` The character to be converted.

Results:

Returns the upper-case equivalent of the given character. If the given character is not a lower-case letter it is returned unchanged.

Errors:

None.

Description:

`toupper` converts the character `c` to its upper-case equivalent. If `c` is not a lower-case letter, it is not converted. Valid lower-case letters are ASCII characters in the range 'a' to 'z'.

See also:

`tolower`

ungetc Pushes a character back onto a file stream.

Synopsis:

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Arguments:

`int c`	The character to be returned.
`FILE *stream`	A pointer to a file stream.

Results:

Returns the pushed back character if successful, or EOF if unsuccessful.

Errors:

Returns EOF if unsuccessful.

Description:

`ungetc` converts `c` to an unsigned char and pushes it back onto the input stream pointed to by `stream`. The next use of any of the `getc` family of functions will return `c` unless a repositioning function has been called in between (**`fflush`**, **`fseek`**, **`rewind`** or **`fsetpos`**).

If `ungetc` is called repeatedly on the same stream without the file stream being read in the meantime, the operation may fail.

`ungetc` is not included in the reduced library.

Example:

```
#include <stdio.h>
#include <ctype.h>

/*
 * Function to read an integer.
 * Leaves the next character to be read
 * as the one immediately after the number.
 */


int get_number()
{
```

```
        int dec = 0;
        int ch;


        while(isdigit(ch = getc(stdin)))
            dec = dec * 10 + ch - '0';
        ungetc(ch,stdin);
        return(dec);
    }
```

See also:

`fflush getc`

`unlink` Deletes a file stream.

Synopsis:

```
#include <iocntrl.h>
int unlink(char *name);
```

Arguments:

`char *name`	The name of the file to be deleted.

Results:

Returns 0 if successful or −1 on error.

Errors:

If an error occurs `unlink` sets `errno` to the value `EIO`.

Description:

`unlink` deletes the file by removing the filename from the host file system. It is equivalent to the ANSI library function `remove`.

`unlink` is not included in the reduced library.

See also:

`remove`

va_arg

Accesses a variable number of arguments in a function definition.

Synopsis:

```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

Arguments:

`va_list ap`	An argument pointer used by the `va_start`, `va_arg` and `va_end` macros.
type	Any C type.

Results:

The first call of `va_arg`, after `va_start`, returns the value of the next parameter in the parameter list after `parmN`. Subsequent calls to `va_arg` return the values of subsequent parameters.

Errors:

If the type specified in `va_arg` disagrees with the type of the next parameter in the parameter list the effects are undefined.

If there is no next argument in the list, or the next argument is a `register` variable, an array type, or a function, the behaviour is undefined. If the next argument is of a type incompatible with the variable type after default promotions, the following compile time error is generated:

```
___assert(0, '"illegal type used with va_arg"')
```

Description:

Each invocation of `va_arg` extracts a single parameter value from a variable length parameter list. `va_arg` must have been initialised by a previous call to `va_start`. The final use of `va_arg` should be followed by a call to `va_end` to ensure a clean termination.

`va_arg` can only be used when there is at least one fixed argument in the variable length parameter list.

`va_arg` is implemented as a macro.

Example:

```
#include <stdio.h>
#include <stdarg.h>

/*
 * Sends the number of strings defined in
   number_of_strings,
 * and given in the parameter list,
   to standard output.
 */

void var_string_print( int number_of_strings, ...)
{
   va_list ap;

   va_start(ap, number_of_strings);
   while (number_of_strings-- > 0)
      puts(va_arg(ap, char *));
   va_end(ap);
}

int main()
{
   var_string_print( 2, "Hello", "World" );

   /*
    * Displays:
    *            Hello
    *            World
    */
}
```

See also:

va_end va_start vfprintf vprintf vsprintf

va_end Cleans up after accessing variable arguments.

Synopsis:

```
#include <stdarg.h>
void va_end(va_list ap);
```

Arguments:

`va_list ap`	An argument pointer used by the `va_start`, `va_arg` and `va_end` macros.

Results:

No value is returned.

Errors:

None.

Description:

`va_end` tidies up after the use of `va_arg`. If it is not used, abnormal function return may occur.

`va_end` can only be used when there is at least one fixed argument in the variable length parameter list.

`va_end` is implemented as a macro.

See also:

`va_arg` `va_start`

va_start

Initialises a pointer to a variable number of function arguments in a function definition.

Synopsis:

```
#include <stdarg.h>

void va_start(va_list ap, parmN);
```

Arguments:

`va_list ap`	An argument pointer used by the `va_start`, `va_arg`, and `va_end` macros.
`parmN`	The name of the last fixed argument in the function definition.

Results:

No value is returned.

Errors:

If `parmN` is declared as storage class `register`, as a function or array, or as a type that is incompatible with the type of the variable after argument promotion, the behaviour is undefined.

Description:

`va_start` is used in conjunction with `va_arg` and `va_end`. It is an initialisation macro for `va_arg`. `va_start` can only be used when there is at least one fixed argument in the variable length parameter list.

`va_start` is implemented as a macro.

See also:

`va_arg` `va_end`

vfprintf An alternative form of `fprintf`.

Synopsis:

```
#include <stdio.h>
int vfprintf(FILE *stream, const char *format,
             va_list arg);
```

Arguments:

`FILE *stream`	An output file stream.
`const char *format`	A format string.
`va_list arg`	A pointer to a list of variable arguments, initialised by `va_start`.

Results:

Returns the number of characters written, or a negative value if an output error occurs.

Errors:

Returns a negative value if an output error occurs.

Description:

`vfprintf` is a form of `fprintf` in which the arguments are replaced by a variable argument list. `vfprintf` should be preceded by a call to `va_start`, and followed by a call to `va_end`.

`vfprinf` is not included in the reduced library.

Example:

```
#include <stdio.h>
#include <stdarg.h>

void write_file(FILE *stream, char *format, ... )
{
   va_list apo;

   va_start(apo,format);
   fputs("WRITE FILE TEXT ", stream);
   vfprintf(stream, format, apo);
   va_end(apo);
```

```
}

int main()
{
 FILE *stream;
 int a = 10;
 char *b = "string";

 stream = fopen("newfile","w");
 if (stream == NULL)
   printf("Error opening file\n");
 else
 {
   write_file(stream, "%d, %s", a, b);
   fclose(stream);
 }
}

/* writes the string "WRITE_FILE TEXT 10, String"
   to the file newfile */
```

See also:

fprintf va_arg va_end va_start vprintf vsprintf

`vprintf` An alternative form of `printf`.

Synopsis:

```
#include <stdio.h>
int vprintf(const char *format, va_list arg);
```

Arguments:

`const char *format`	A format string
`va_list arg`	A pointer to a list of variable arguments, initialised by `va_start`.

Results:

Returns the number of characters written, or a negative value if an output error occurred.

Errors:

`vprintf` returns a negative value if an output error occurs.

Description:

`vprintf` is a form of `printf` in which the arguments are replaced by a variable argument list. `vprintf` should be preceded by a call to `va_start`, and followed by a call to `va_end`.

`vprinf` is not included in the reduced library.

See also:

`printf va_arg va_start va_end vfprintf vsprintf`

vsprintf

An alternative form of `sprintf`.

Synopsis:

```
#include <stdio.h>
int vsprintf(char *s, const char *format,
             va_list arg);
```

Arguments:

`const char *s`	The string to which the formatted string is written.
`const char *format`	A format string.
`va_list arg`	A pointer to a list of variable arguments, initialised by `va_start`.

Results:

Returns the number of characters written.

Errors:

None.

Description:

`vsprintf` is a form of `sprintf` in which the arguments are replaced by a variable argument list. `vsprintf` should be preceded by a call to `va_start`, and followed by a call to `va_end`.

To use `vsprintf` in the reduced library include the header file `stdiored.h`.

See also:

`sprintf` `vfprintf` `vsprintf` `va_arg` `va_end` `va_start`

write Writes bytes to a file stream. File handling primitive.

Synopsis:

```
#include <iocntrl.h>
int write(int fd, char *buf, int n);
```

Arguments:

`int fd`	A file descriptor.
`char *buf`	A pointer to a buffer from which the bytes are obtained.
`int n`	The maximum number of bytes that `write` will attempt to output.

Results:

Returns the number of bytes written or −1 on error.

Errors:

If an error occurs `write` sets `errno` to the value `EIO`.

Description:

`write` writes n bytes from the buffer pointed to by `buf` to the file specified by `fd`. If `n` is zero or negative no output occurs.

`write` is not included in the reduced library.

See also:

`read`

Language Reference

72 TDS 225 00

August 1990

3 New features in ANSI C

This appendix describes the new features added by the ANSI standard to the C language.

This chapter is not intended to be a reference to ANSI standard C but rather a summary of differences from the previous widely-known definition of the language. For a formal description of the language the reader is referred to the ANSI reference documents and to '*C: A Reference Manual*' by Harbison and Steel.

Kernighan and Ritchie's original description of the language as defined in their book '*The C programming language*' (First edition 1978), is referred to in this chapter as 'K & R C'.

Details of these publications can be found in the bibliography to the rear of this manual.

This chapter is divided into two sections:

3.1 A summary of the new features added by ANSI to the original definition of the language.

3.2 Detailed descriptions of the new features.

3.1 Summary of new features in the ANSI standard

The following tables list the new features in the ANSI standard. The tables list the main areas of change and briefly describe how they differ from the original implementation of the language.

Area of change	ANSI standard
Function decls.	Parameter lists in function declarations can include type specifiers with or without identifiers. The new `void` type can be used and the list may end with an ellipsis '`...`' to indicate a variable number of parameters.

Type Specifiers	1. New types:
	`enum`
	`void`
	2. New type qualifiers:
	`const`
	`signed`
	`volatile`
	Where specified alone, `signed`, `const`, and `volatile` imply the appropriately qualified `int` type.
	3. New unsigned types:
	`unsigned char`
	`unsigned long`
	`signed char`

Identifiers	The first 31 characters of internal names are significant.

Keywords	1. Keyword `entry` is no longer valid.
	2. New keywords:
	`const`
	`enum`
	`signed`
	`void`
	`volatile`

Area of change	ANSI standard
Constants	Integer constants can use the suffix `U` to denote an unsigned integer constant.
	Floating point constants can use the suffixes `F` (for `float`) and `L` (for `long double`).
Operators	New unary operator '+' added to complement '–'.
Character types	Character constants are of type `int` and are sign extended in type conversions.
	New character escape codes: `\"` `\?` `\x` `\a` `\v`.
	Signedness of `char` types is implementation defined.
Hardware characteristics	The type `short` is at least 16 bits long and the type `long` at least 32 bits long.
Compiler control lines	New preprocessor directives: `#elif` `#error` `#pragma`
	Some preprocessor macros are also defined.
Structures and unions	Structures or unions can be:
	Assigned to other structures or unions. Passed by value to functions. Returned by functions.
Initialisation	Unions can be initialised.
Trigraphs	Character trigraphs are introduced to support the ISO 646 invariant character set.

3.2 Details of new features

3.2.1 Function declarations

A new form of function declaration is available which allows types to be specified for parameters in the function's parameter list. Declarations can omit parameter identifiers and give only the type specifiers.

It is also possible to specify a variable number of parameters by terminating the parameter list with an ellipsis '...'. For example:

```
void add_numbers(int *sum, int a, int b);
        /* Declaration with identifiers */

void add_numbers(int *, int, int);
        /* Declaration without identifiers */

void add_many_numbers(int *sum, int n, ...);
        /* Declaration with variable parameters */
```

A function with no parameters can be specified by specifying the keyword `void` as the only parameter in the parameter list. For example:

```
int hello(void);
```

A function declarator using a parameter type list defines a prototype for that function.

3.2.2 Function prototypes

Function prototypes are a new way of declaring functions. They make programs easier to read and function call errors easier to find.

When using function prototypes:

1 Functions must be explicitly declared before any call is made.

2 Multiple declarations of the same function must agree exactly.

3 Function declarations must use the parameter type list form.

4 When calling a function, the number and types of the parameters must agree with the specification in the declaration.

5 Arguments to functions are converted to the types specified in the declaration.

3.2.3 Declarations

Type specifiers can be used in pointer declarations. This is particularly useful for creating constant pointers, pointers to constants and pointers to volatiles. For example:

```
const int *ptr_to_constant;
       /* Declares a pointer to a constant int  */

int *const constant_ptr;
       /* Declares a constant pointer to an int */

volatile int *ptr_to_volatile;
       /* Declares a pointer to a volatile int  */
```

3.2.4 Types and type qualifiers

This section describes the ANSI standard syntax for types and type specifiers.

The following type specifiers have been added: **`const enum signed void volatile`**.

`const` defines a constant object cannot be changed in the program. **`const`** can be used alone or with other type specifiers **`struct union enum`** and **`volatile`**. Used alone it implies **`const int`**. For example:

```
const int month = 10;

month = 11; /* Not allowed */
month++;    /* Not allowed */
```

`const` can be used within pointer declarations to declare variable pointers to constant values, or constant pointers to variable values.

`enum` is used to create enumerated types. An enumerated type defines a sequence of integer values for groups of logical names. The sequence of values begins at 0 and increments by one unless specific values are assigned. For example:

```
/* Define an enumerated type for the days of
                                   the week */
enum days {monday, tuesday, wednesday, thursday,
                      friday, saturday, sunday};
enum days today; /* Declare today as a variable
                                of type days */
```

```
today = friday;
if (today == sunday)
.
.
.
```

The default value of a constant can be overridden by assigning a a specific integer value. If a member of the list is not assigned a value explicitly, it takes on the value of (previous constant + 1). For example:

```
enum poets {corso, burroughs, ginsberg = 9, cummings};
/* corso = 0, burroughs = 1, cummings = 10 */
```

`signed` complements the existing type specifier `unsigned`. It may be used alone, where it implies `signed int`, or to qualify the following types: `int short int long int char`.

`void` is mainly used to declare functions which do not return a value. For example:

```
void add_numbers();
main()
{
int *answer;
add_numbers(answer,23,42);
}
.
.
.
void add_numbers(sum, b, c)
int *sum;
int b,c;
{
  sum = b + c;
}
```

Another use for `void` is in a cast expression where a returned value is discarded. For example:

```
/* Ignore the return value of fputc */
(void) fputc(ch,stream);
```

`volatile` identifies an object as modifiable outside the control of the implementation. For example, the object may refer to a memory mapped port which is used by a modem. `volatile` can be used to protect objects from unpredictable compiler optimizations.

`volatile` can be used alone or with other type specifiers. used alone `volatile` implies `volatile int`.

An object can be both `volatile` and `const` in which case it can not be modified by the program but could be modified by an external process (for example, a real time clock). For example:

```
volatile int port_one;
const volatile int clock;
```

3.2.5 Constants

This section summarises the changes to the syntax for integer, floating point, string and character constants.

The suffix `U` can follow integer constants to indicate type `unsigned`. `U` can be used in conjunction with the existing `L` suffix and the order is not significant. For example:

```
42u   1096U   1001u   2048UL
```

The suffix `F` can follow floating point constants to indicate type `float` and the suffix `L` to indicate type `long double`. For example:

```
3.1F    4.2L
```

The type `long float` is no longer allowed.

Adjacent string constants are concatenated into a single string terminated by a null. The following new character escape codes are defined:

Code	Description
`\?`	Gives the question mark character. This should be used where a question mark could be mistaken for part of a trigraph.
`\"`	Gives the double quote character.
`\a`	Rings the bell (equivalent to CTRL-G).
`\v`	Gives a vertical tab.
`\x`*n*	Gives the character represented by *n*, where *n* is the ASCII code of the character represented in hexadecimal. For example, `\x2B` gives the character +.

3.2.6 Preprocessor extensions

This section describes the predefined preprocessor directives and macros.

Compiler directives

Directive	Description
`#elif`	Abbreviation of `#else #if`.
`#error`	Generates a compiler error message containing optional text.
`#pragma`	Causes an implementation-defined effect. In ANSI C this directive is used to select a particular combination of compiler options or to override options given on the command line.

Predefined macros:

Macro	Description
`__DATE__`	The current date, in the form: `Mmm dd yyyy`.
`__FILE__`	The name of the current source file, expressed as a string literal.
`__LINE__`	The line number of the current line in the source file, expressed as a decimal constant.
`__STDC__`	A non-zero value if the implementation conforms to ANSI C.
`__TIME__`	The current time, in the form: `hh:mm:ss`.

3.2.7 Structures and unions

In ANSI C structures and unions can be assigned to other structures or unions, passed by value to functions, and returned by functions. Unions can be initialised.

When a structure is given as an argument to a function a copy of the structure is created for use within the function. For example:

```
struct record
{
  char firstname[30];
  int  age;
};

void print_name(struct record person);

struct record test(struct record first,
                   struct record second);
```

```
main()
{
  struct record ph;
  struct record rl;

  ph.firstname = "Phil";
  ph.age   = 27;

  /* Assigning a structure to a structure */
  current_person = ph;

  /* Passing a structure as an argument to a
                                        function */
  print_name(current_person);

  /* Returning a structure from a
                             function */
  winner = test(ph, rl);
}
```

Unions can be initialised. The initialisation is performed according to the type of its first component and the expression used to perform the initialisation must evaluate to the correct type. For example:

```
union alltypes {
  double bigfloat;
  int digit;
  char letter;
} initalltypes = 3.1;

union complex {
  struct {int a; char b;} s;
  double bigfloat;
} initcomplex = {42, 'x' };
```

3.2.8 Trigraphs

Trigraphs are added to enable C programs to be written using only the ISO 646 invariant code set. ISO 646 is a subset of 7-bit ASCII which contains only those characters present on all keyboards.

Trigraphs and the characters that they represent are listed in the following table.

Trigraph	Character represented
??=	#
??(	[
??)	]
??/	\
??'	^
??<	{
??>	}
??!	|
??-	~

All other trigraph-like sequences are treated as literal strings. For example, the sequence ??+ is not a trigraph and is treated as the literal sequence that it represents.

Trigraphs are converted to the equivalent character before lexical analysis takes place.

Trigraph escape codes

The character escape code \? has been added to allow the printing of trigraph strings. The trigraph string should be preceded by the escape character. For example:

```
static char texta[] = "This is a backslash: ??/";
static char textb[] = "This is not a trigraph \??/";
```

4 Language extensions

This appendix summarises the INMOS extensions to the C language. It describes the concurrency features, compiler pragmas, and lists the predefinitions, all of which are described in detail elsewhere in this book, It also describes the `__asm` statement that supports the insertion of transputer code into C programs.

The INMOS implementation of ANSI C provides the following language extensions beyond the ANSI standard:

- Concurrency support.
- Pragmas.
- Additional predefined macros.
- Assembly language support.

4.1 Concurrency support

Concurrency support is provided by a set of library functions with associated predefined data types and data structures. The library functions are declared in three standard C header files along with all related constants and macros.

Functions are provided for creating and manipulating processes (**process.h**), for synchronising processes and exchanging data down channels (**channel.h**), and for creating and manipulating semaphores (**semaphore.h**).

Full details of how to create parallel programs using the ANSI C concurrency extensions can be found in chapter 4 '*Parallel processing*' of the accompanying User Manual.

4.2 Pragmas

A series of special compiler operations are implemented as options to the **#pragma** directive. The options available are listed below. Details of the pragmas, their syntax and options can be found in section 11.3.1 in the accompanying User Manual.

Pragma	Description
`IMS_on`	Enables specific compiler checks. Checks to be enabled are specified as arguments to the pragma.
`IMS_off`	Disables specific compiler checks. Takes the same set of check arguments as `IMS_on`.
`IMS_linkage`	Adds tags for segment ordering.
`IMS_nolink`	Enables functions to be compiled without a static link parameter. Used when calling occam code from C, and C functions from occam.
`IMS_codepatchsize`	Notifies the linker of a reserved code patch and specifies its size.
`IMS_modpatchsize`	Notifies the linker of a reserved module number patch and specifies its size.
`IMS_translate`	Translates all references to one name into another name. Used to create aliases for external routines which contain prohibited characters.

4.3 Predefined macros

The following predefined macros are provided in the ANSI C toolset in addition to the standard definitions required by the ANSI standard.

Constant	Meaning/value
`__CC_NORCROFT`	Indicates a compiler derived from the Norcroft C compiler. Set to the decimal constant one (1).
`_ICC`	Indicates the ANSI C compiler `icc`. Set to the decimal constant one (1).
`_PTYPE`	Indicates the target processor type. Takes the following values: 2 – T212 3 – T225 4 – T414 5 – T425/T400 8 – T800 9 – T801/T805 A – Class TA B – Class TB
`_ERRORMODE`	A decimal constant indicating the execution error mode. Takes the following values: 1 – HALT 2 – STOP 3 – UNIVERSAL
All compiled object code generated by `icc` is in UNIVERSAL mode.	

4.4 Assembly language support

The insertion of transputer code into C programs is performed using the `__asm` statement. Sequences of transputer instructions specified in this way are assembled in line by the compiler.

The rest of this section assumes some familiarity with the transputer instruction set. For a list of transputer instructions see appendix B '*Transputer instruction set*' in the accompanying User Manual.

A more detailed description of the instruction set including information about architecture and design can be found in '*Transputer instruction set: a compiler writer's guide*'.

The full syntax of the `__asm` statement is given in section A.3.

4.4.1 Directives and operations

`__asm` statements can contain any number of primary or secondary transputer operations, optionally preceded by a **`size`** qualifier, or transputer pseudo-operations. Any transputer instruction can be prefixed with a label.

In the transputer instruction set primary operations are *direct* functions, *prefixing* functions, or the special indirect function *opr*. Primary operations are always followed by an operand which can be any constant or constant expression. If additional **`pfix`** and **`nfix`** instructions are required to encode large values the assembler automatically generates the required bytes.

Secondary operations are any transputer *operation*, that is, any instruction selected using the *opr* function.

Pseudo-operations are more complex operations built up from sequences of instructions. Like macros, they expand into one or more transputer instructions, depending on their context and parameters.

Pseudo-operations that are supported by `__asm` are listed below. A full syntax

definition for pseudo-operations can be found in section A.3.

`ld`	*expression*
`st`	*lvalue*
`ldab`	*expression*, *expression*
`stab`	*lvalue*, *lvalue*
`ldabc`	*expression*, *expression*, *expression*
`stabc`	*lvalue*, *lvalue* , *lvalue*
[`size` *constant*]	`j` *label*
[`size` *constant*]	`cj` *label*
[`size` *constant*]	`call` *label*
[`size` *constant*]	`ldlabeldiff` *label* - *label*
`byte`	*constant* { , *constant* }
`word`	*constant* { , *constant* }
`align`	

*lvalue*s can be any valid C expression, and labels can be any valid C identifier. The load and store pseudo-ops (`ld`, `st`, `ldab`, `stab`, `ldabc`, `stabc`) load or store the integer registers **Areg**, **Breg**, and **Creg**.

The `ldlabeldiff` operation loads the difference between the addresses of two labels into **Areg**.

4.4.2 `size` option

The `size` option on primary operations, secondary operations, and certain pseudo-operations, forces the instruction to occupy a set number of bytes. If the instruction is shorter than this it is padded out with trailing prefix 0 instructions. If the instruction cannot fit in the specified number of bytes, an error is reported. The `size` option allows instructions to be built of the same size and is intended to assist with the creation of jump tables.

4.4.3 Labels

Labels can be placed on `__asm` statements or on any line of transputer code. Labels placed inside and outside the `__asm` statement are handled identically. C statements are permitted to `goto` a label set inside an `__asm` statement and vice versa.

4.4.4 Notes on transputer code programming

1 Floating-point (fp) registers cannot be loaded directly; they must be loaded or stored by first loading a pointer to the register into an integer register and then using the appropriate floating-point load or store instruction.

2 The operands to the load pseudo-ops must be small enough to fit in a register and the operands to the store pseudo-ops must be word-sized modifiable *lvalues*.

3 Only the lower eight bits of the *constant* operand(s) of the `byte` pseudo-op are generated.

4 The `word` pseudo-op generates word-length constants for the target machine. If a constant is too large to fit in the machine's word length only the lower bits are generated.

5 The `align` pseudo-op generates padding bytes (prefix 0) until the current code address is on a word boundary.

4.4.5 Useful predefined variables

The following variables are predefined in the compiler and may be used in expressions as though they were user-defined variables:

`volatile const void *_lsb`	Pointer to the base of a file's static area.
`volatile const void *_params`	Pointer to the base of the current function's parameter block.

Given access to a function's parameter block, it is possible to determine the function's return address, the global static pointer, and the calling function's workspace as in the following example:

```
void p(int a, int b)
{
  typedef struct paramblock
      { void *return_address;
        void *gsb;
        int regparam1, regparam2;
      } paramblock;

  volatile const void *_params;

  paramblock *pp = _params;
  /*  return address is:     pp->return_address
```

```
        global static base is: pp->gsb
        caller's wptr is:      (void *)(pp + 1); */
}
```

4.4.6 Transputer code examples

This section contains listings of programs fragments that illustrate common uses of embedded instruction code.

Setting the transputer error flag

```
void set_error_flag(void)
{
  __asm { seterr; }
}
```

Loading constants using literal operands

```
#define answer 42
const int c
__asm {
   ldc   17;         /* decimal */
   ldc   0xff;       /* hex */
   ldc   0377;       /* octal */
   ldc   answer;     /* defined by macro */
   ldc   sizeof(c); /* constant expression */
   ldc   10+7;       /*      ditto          */
}
```

Labels and jumps

```
void p(void)
{
  int a, b, c;
  /* The following code performs
     if (b > c) a = b; else a = c; */
  __asm{
        ld    b;
        ld    c;
        gt;
        cj    label1;
        ld    b;
        st    a;
        j     done;
```

```
label1:
        ld    c;
        st    a;
done:   ;
  }
}
```

Jump tables

```
#include <stdio.h>
#define JUMP_SIZE 3
void p(int i)
{
  __asm{ ld          i;
            /* load the index */
        adc          -1;
            /* subtract base subscript */
        ldc          JUMP_SIZE;
            /* scale by size of table entry */
        prod;
        ldlabeldiff table - here;
            /* load pointer to start of table */
        ldpi;
here:
        bsub;
            /* add the offset */
        gcall;
            /* jump to ith. entry */

table:
        size JUMP_SIZE j lab1;
        size JUMP_SIZE j lab2;
        size JUMP_SIZE j lab3;
        size JUMP_SIZE j lab4;
      }
  lab1: printf("i = 1"); return;
  lab2: printf("i = 2"); return;
  lab3: printf("i = 3"); return;
  lab4: printf("i = 4"); return;
}
```

Loading floating point registers

```
void p(void)
{
  float a, b, c;
  /* The following code performs
```

```
          a = b - c;               */
  __asm{
         ld     &b;
         fpldnlsn;
         ld     &c;
         fpldnlsn;
         fpsub;
         ld     &a;
         fpstnlsn;
       }
}
```

Using align/word to return an element of a table

```
int p(int i)
{
  /* The following code returns the ith
  /* element of the table defined below */

  int res;
  __asm{
   ld           i;
   ldlabeldiff table - here;
   ldpi;
here:
         wsub;
         ldnl        0;
         st          res;
         j           done;
   /* Make sure table is word aligned
         /* for ldnl to work correctly */
   align;
table:
         word        1, 1, 2, 3, 5, 8, 13, 21, 34;
       }
done:
  return res;
}
```

Inserting raw machine code

The following code inserts the actual machine code (in hex) for the *ret* instruction.

```
void ret_hex(void)

{
```

```
    __asm { byte 0x22, 0xF0; }
}
```

5 Implementation details

This appendix describes the implementation of the language in areas where the ANSI standard is flexible or allows alternative solutions.

5.1 Data type representation

5.1.1 Scalar types

C scalar type representations on 32 and 16 bit transputers are described in the following table.

`char`, `unsigned char`	32	Represented in a word in which the lower eight bits are significant, the upper bits are zero.
	16	Same as 32 bit.
`signed char`	32	Represented in a word in which the lower eight bits are significant, bit 7 is the sign-bit, the upper bits are zero.
	16	Same as 32 bit.
`unsigned short`	32	Represented in a word in which the lower 16 bits are significant, the upper bits are zero.
	16	Represented in a word in which all 16 bits are significant.
`signed short`	32	Represented in a word in which the lower 16 bits are significant, bit 15 is the sign bit, the upper bits are zero.
	16	Represented in a word in which all 16 bits are significant, bit 15 is the sign bit.
`unsigned int`	32	Represented in a word in which all 32 bits are significant.
	16	Represented in a word in which all 16 bits are significant.
`signed int`	32	Represented in a word in which all 32 bits are significant, bit 31 is the sign bit.
	16	Represented in a word in which all 16 bits are significant, bit 15 is the sign bit.

`unsigned long`	32	Represented in a word in which all 32 bits are significant.
	16	Represented in two words in which all 32 bits are significant, the lower addressed word contains the least significant bits.
`signed long`	32	Represented in a word in which all 32 bits are significant, bit 31 is the sign bit.
	16	Represented in two words in which all 32 bits are significant, bit 31 is the sign bit. The lower addressed word contains the least significant bit.
`float`	32	Represented in a word, in IEEE single-precision format.
	16	Represented in two words, in IEEE single-precision format.
`double`	32	Represented in two words, in IEEE double-precision format.
	16	Represented in four words, in IEEE double-precision format.
`enumeration`	32	Represented in a word in which all 32 bits are significant.
	16	Represented in a word in which all 16 bits are significant.

All `signed` integer types are represented in twos-complement form and all `unsigned` integer types in binary form.

All floating point types are represented in a form defined by the ANSI/IEEE standard 754-1985.

5.1.2 Arrays

Each element of an array of `char` occupies 8 bits and each element of an array of `short` occupies 16 bits.

Elements of arrays of any other type are represented as the element would be represented if it was not in an array. An array is padded at the high-end address to the next word boundary: the padding has no defined value.

5.1.3 Structures

Structure fields are allocated starting from the lowest address. Fields of type `char` are allocated on a byte boundary, and are represented in 8 bits.

On 32-bit machines only, fields of type `short` are allocated on an even-address boundary, and are represented in 16 bits. Thus, adjacent `char` or `short` fields may be packed into the same word.

Adjacent bit-fields are packed into the same word if possible: the first bit-field is placed in the least significant bits of the word. If there is not enough room left after a previous bit-field, a bit-field will be placed in the least significant bits of the next word. Fields of any other type are represented as they would be if the field was not in a structure. A structure is padded at the high-end address to the next word boundary: the padding has no defined value.

5.1.4 Unions

Each field of a union is represented as it would be if it was not in a union. A union is padded at the high-end address to the next word boundary: the padding has no defined value.

5.2 Type conversions

5.2.1 Integers

The result of converting an unsigned integer, u, to a signed integer, s, of equal length, if the value cannot be represented, is calculated as follows:

If $max.s$ is the largest number that can be represented in the signed type then:

$$result = u - 2(max.s + 1)$$

An integer is converted to a shorter signed integer, by first converting it to an unsigned integer of the same length as the shorter signed integer (by taking the nonnegative remainder on division by the number one greater than the largest unsigned number that can be represented in the type with smaller size), and then converting to the corresponding signed integer, as described above.

5.2.2 Floating point

When converting an integral number to a floating-point number that cannot exactly represent the original value, the IEEE 754 'Round to Nearest' rounding mode is used.

When converting a floating-point number to a narrower floating-point number, the IEEE 754 'Round to Nearest' rounding mode is used.

5.3 Compiler diagnostics

Diagnostics are generated at four severity levels: *Warning*; *Error*; *Serious*; and *Fatal*. All compiler messages are generated in standard toolset format (see section A.6 in the accompanying user manual).

5.4 Environment

5.4.1 Arguments to `main`

The interface to `main` is as follows:

```
int main(int argc, char *argv[], char *envp[],
         Channel *in[], int inlen,
         Channel *out[], int outlen);
```

where: `int argc` is the number of arguments passed to the program from the environment, including the program name.

`char *argv[]` is an array of pointers to the passed arguments.

`char *envp[]` is an array of pointers for the `getenv` function. In this implementation it is set to NULL.

`Channel *in[]` is an array of input channels.

`int inlen` is the size of the input channel array.

`Channel *out[]` is an array of output channels.

`int outlen` is the size of the output channel array.

The first two input and output channels are reserved; `in[1]` is the channel coming from the server, `out[1]` is the channel going to the server.

`in[0]` and `out[0]` are unused.

5.4.2 Interactive devices

`stdin`, `stdout` and `stderr` are treated as if they are connected to an interactive device.

5.5 Identifiers

The 255 initial characters (beyond 31) in an identifier without external linkage, and the 255 all initial characters (beyond 6) in an identifier with external linkage, are significant.

Case distinctions are significant in an identifier with external linkage.

5.6 Source and execution character sets

The source character set comprises those characters explicitly specified in the Standard, together with all other printable ASCII characters. The execution character set comprises all 256 values 0 - 255. Values 0 - 127 represent the ASCII character set.

There are eight bits in a character in the execution character set.

Each member of the source character set is a member of the ASCII character set and maps to the same member of the ASCII character set in the execution character set.

All characters and wide characters are represented in the basic execution characters set. The escape sequences not represented in the basic execution character set are the octal integer and hexadecimal integer escape sequences, whose values are defined by the Standard.

Shift states for encoding multibyte characters

There is only one shift state, which is the initial shift state as specified in the Standard. Multibyte characters do not alter the shift state.

Integer character constants

The value of an integer character constant that contains more than one character is given by:

$$\sum_i (value\ of\ ith\ character << (8 * i))$$

Wide character constants which contain more than one multibyte character are disallowed.

Locale used to convert multibyte characters

The only locale supported to convert multibyte characters into corresponding wide characters (codes) for a wide character constant is the 'C' locale.

Plain chars

A "plain" char has the same range of values as `unsigned char`.

5.7 Integer operations

Bitwise operations on signed integers

Signed integers are represented in twos complement form. The bitwise operations operate on this twos complement representation.

Sign of the remainder on integer division

The remainder on integer division takes the same sign as the divisor.

Right shifts on negative-valued signed integral types

Signed integers are represented in twos complement form. The right-shift operates on this twos complement form; zero bits are shifted in at the left-hand side; thus a negative-valued signed integer, if right-shifted more than zero places, will become positive.

5.8 Registers

The compiler attempts to `register` variables at shorter offsets from the workspace pointer.

5.9 Enumeration types

The values of enumeration types are represented as `ints`.

5.10 Bit fields

A "plain" int bit-field is treated as an `unsigned int` bit-field.

Bit-fields are allocated low-order to high-order within an `int` (i.e. the first field

textually is placed in lower bits in the `int`).

A bit-field cannot straddle a word boundary.

5.11 `volatile` qualifier

An access to an object that has volatile-qualified type is a 'read' from the memory location containing the object (if the object's value is required), or a 'write' to the memory location containing the object (if the object is assigned to).

If the volatile object is an array, then the access will be only to the appropriate element of the array.

If the volatile object is a structure and only a field of the structure is required, then the access will be only to the appropriate field.

If the object is not an array element or structure field, then the object occupies a whole number of words, and all the words will be accessed. Otherwise, if the array element or structure field is shorter than a word, then only the appropriate bytes will be accessed.

If the object is a bit-field, then in the case of read access, the entire word containing the bit-field will be read; and in the case of write access, the entire word containing the bit-field will be first read, and then written.

Note: If the object is an array element or structure field of type `short` on a 32-bit transputer, or if the object is larger than two words, then the transputer block move instruction is used for the access. On some transputers, if a block move instruction is interrupted, when it resumes it may reread the same word of memory which was read immediately before the interrupt. This may cause problems with some peripheral devices.

5.12 Declarators

There is no restriction upon the number of declarators that may modify an arithmetic, structure, or union type.

5.13 Switch statement

There is no restriction upon the number of case values in a switch statement.

5.14 Preprocessing directives

Constants controlling conditional inclusion

The value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Such a character constant may NOT have a negative value.

Date and time defaults

When date of translation is not available, `__DATE__` expands to

```
"Jan 1 1900"
```

When time of translation is not available, `__TIME__` expands to

```
"00:00:00"
```

5.15 Runtime library

The null pointer constant to which the macro `NULL` expands to is `(void *)0`.

Appendices

A Syntax of language extensions

This appendix defines the language extensions in the ANSI C toolset.

A.1 Notation

Syntax definitions are presented in a modified Backus-Naur Form (BNF). Briefly:

1 Terminal strings of the language – those not built up by rules of the language – are printed in teletype font e.g. `void`.

2 Each phrase definition is built up using a double colon and an equals sign to separate the two sides.

3 Alternatives are separated by vertical bars ('|').

4 Optional sequences are enclosed in square brackets ('[' and ']').

5 Items which may be repeated zero or more times appear in braces ('{' and '}').

A.2 `#pragma` directive

```
control-line   ::=  #pragma pragma (params)

pragma         ::=    IMS_on  (params)
                    | IMS_off  (params)
                    | IMS_linkage  (["name"])
                    | IMS_nolink (functionname)
                    | IMS_modpatchsize(n)
                    | IMS_codepatchsize(n)
                    | IMS_translate  (name,  "newname")
```

```
params        ::=   channel_pointers    | cp
                  | inline_ops          | il
                  | inline_string_ops   | is
                  | printf_checking     | pc
                  | scanf_checking      | sc
                  | stack_checking      | sc
                  | warn_bad_target     | wt
                  | warn_deprecated     | wd
                  | warn_implicit       | wi
```

A.3 __asm statement

```
asm-statement ::=   __asm { asm-directive }

asm-directive ::=   [ size constant ] primary-op constant ;
                  | [ size constant ] secondary-op ;
                  | pseudo-op ;
                  | identifier: asm-directive
                  | ;

pseudo-op     ::=   ld expression
                  | st lvalue
                  | ldab expression , expression
                  | stab lvalue , lvalue
                  | ldabc expression , expression , expression
                  | stabc lvalue , lvalue , lvalue
                  | [ size constant ] j label
                  | [ size constant ] cj label
                  | [ size constant ] call label
                  | [ size constant ] ldlabeldiff label - label
                  | byte constant { , constant }
                  | word constant { , constant }
                  | align
```

B ANSI compliance data

This appendix lists details of the INMOS implementation of C in areas of the language where formal documentation is required by the ANSI standard. The information is provided for compliance with the standard and to provide a convenient reference point for programmers wishing to port the toolset to other hosts.

The formal ANSI requirement in each area is given followed by a reference to the appropriate section in the standards document. This is followed by a description of the INMOS implementation in that area.

Where the information required is provided in other areas of this book or in the companion volume the '*ANSI C toolset user manual*' a reference is given to the appropriate section.

B.1 Translation

- **How a diagnostic is identified (§2.1.1.3)**
  Diagnostics are displayed to `stderr` (UNIX and VMS) or `stdout` (MS-DOS) in a standard format. The display format is described in section A.6 the accompanying user manual.

B.2 Environment

- **The semantics of the arguments to main (§2.1.2.2)**
  The prototype of C `main` is as follows:

```
int main (int argc, char *argv[], char *envp[],
          Channel *in[], int inlen,
          Channel *out[], int outlen);
```

where: `argc` is the number of arguments passed to the program from the environment, including the program name.

`*argv` is an array of pointers to those arguments.

`*envp` is an array of pointers for the `getenv` library function – implemented in ANSI C as NULL.

`Channel *in[]` is an array of input arguments.

`int inlen` is the size of the array.

`Channel *out[]` is an array of output arguments.

`int outlen` is the size of the array.

An extension for configured programs allows extra parameters to be passed by defining them as `interface` parameters within the configuration description. These configuration level parameters can be accessed by the C program using the runtime library function `*get_param`.

- **What constitutes an interactive device (§2.1.2.3)**
  `stdin`, `stdout` and `stderr` are treated as if they are connected to an interactive device.

B.3 Identifiers

- **The number of significant initial characters (beyond 31) in an identifier without external linkage (§3.1.2).**
  The first 255 characters in the identifier are significant.

- **The number of significant initial characters (beyond 6) in an identifier with external linkage (§3.1.2).**
  The first 255 characters in the identifier are significant.

- **Whether case distinctions are significant in an identifier with external linkage (§3.1.2).**
  Case distinctions are significant in an identifier with external linkage.

B.4 Characters

- **The members of the source and execution character sets, except as explicitly specified in the Standard (§3.2.1).**
  The source character set comprises those characters explicitly specified in the Standard, together with all other printable ASCII characters. The execution character set comprises all 256 values 0 - 255. Values 0 - 127 represent the ASCII character set.

- **The shift states used for the encoding of multibyte characters (§2.2.1.2).** There is only one shift state, which is the initial shift state as specified in the Standard. Multibyte characters do not alter the shift state.

- **The number of bits in a character in the execution character set (§2.2.4.2).** There are eight bits in a character in the execution character set.

- **The mapping of members of the source character set (in character constants and string literals) to members of the execution character**

set (§3.1.3.4).
Each member of the source character set is a member of the ASCII character set. It maps to the same member of the ASCII character set in the execution character set.

- **The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant (§3.1.3.4).**
  All characters and wide characters are represented in the basic execution character set.

  The escape sequences not represented in the basic execution character set are the octal integer and hexadecimal integer escape sequences, whose values are defined by the Standard.

- **The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character (§3.1.3.4).**
  See section 5.6.

- **The current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant (§3.1.3.4).**
  The only locale supported is the 'C' locale.

- **Whether a "plain" char has the same range of values as signed char or unsigned char.**
  A "plain" char has the same range of values as unsigned char.

B.5 Integers

- **The representations and sets of values of the various types of integers (§3.1.2.5).**
  For all data-type representations see section 5.1.1 in this manual.

- **The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented (§3.2.1.2).**
  See section 5.2.1.

- **The results of bitwise operations on signed integers (§3.3).**
  Signed integers are represented in twos complement form. The bitwise operations operate on this twos complement representation.

- **The sign of the remainder on integer division (§3.3.5).**

The remainder on integer division takes the same sign as the divisor.

- **The result of a right shift of a negative-valued signed integral type (§3.3.7).**
Signed integers are represented in twos complement form. The right-shift operates on this twos complement form; zero bits are shifted in at the left-hand side; thus a negative-valued signed integer, if right-shifted more than zero places, will become positive.

B.6 Floating point

- **The representations and sets of values of the various types of floating-point numbers (§3.1.2.5).**
For all data-type representations see section 5.1.1 in this manual.

- **The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value (§3.2.1.3).**
When converting an integral number to a floating-point number, the IEEE 754 'Round to Nearest' rounding mode is used.

- **The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number (§3.2.1.4).**
When converting a floating-point number to a narrower floating-point number, the IEEE 754 'Round to Nearest' rounding mode is used.

B.7 Arrays and pointers

- **The type of integer required to hold the maximum size of an array, that is, the type of the sizeof operator, `size_t` (§3.3.3.4, §4.1.1).**
The type of the sizeof operator, `size_t`, is unsigned int.

- **The result of casting a pointer to an integer or vice versa (§3.3.4).**
When a pointer is cast to an integer, the bit representation remains unchanged.

  N.B. A NULL pointer on a 32-bit transputer has the representation all bits zero, so that casting an integer variable of value zero to a pointer will result in a NULL pointer. However, a NULL pointer on a 16-bit transputer DOES NOT have the representation all bits zero, so that it is incorrect to assume that an integer *variable* of value zero, when cast to a pointer will result in a NULL pointer. (the ANSI standard guarantees that an integer *constant* of value zero, when cast to a pointer, will result in a NULL pointer.)

- **The type of integer required to hold the difference between two pointers to elements of the same array, `ptrdiff_t` (§3.3.6, §4.1.1).**
  `int`.

  Note that this means that it is not possible to declare an array of `char`-sized objects which is larger than half of the integer range, and take the difference of a pointer to the end and a pointer to the start. This is particularly important on a 16-bit processor, ie. `ptrdiff_t` will not correctly represent the difference between the two ends of an array of `char`-sized objects larger than 32767 bytes.

  There is no problem with arrays of elements which are larger than `char`.

B.8 Registers

- **The extent to which objects can actually be placed in registers by use of the `register` storage-class specifier (§3.5.1).**
  The `register` storage class specifier is used to allocate objects at a lower offset in workspace. Objects cannot be placed in registers.

B.9 Structures, unions, enumerations, and bit-fields

- **A member of a union object is accessed using a member of a different type (§3.3.2.3).**
  For the implementation of unions see section 5.1.4 in this manual.

- **The padding and alignment of members of structures (§3.5.2.1). This should present no problem unless binary data written by one implementation are read by another.**
  For the implementation of structures see section 5.1.3 in this manual.

- **Whether a "plain" int bit-field is treated as a signed int bit-field or as an unsigned int bit-field (§3.5.2.1).**
  A "plain" int bit-field is treated as an unsigned int bit-field.

- **The order of allocation of bit-fields within an int (§3.5.2.1).**
  Bit-fields are allocated low-order to high-order within an int (ie. the first field textually is placed in lower bits in the int).

- **Whether a bit-field can straddle a storage-unit boundary (§3.5.2.1).**
  A bit-field cannot straddle a word boundary.

- **The integer type chosen to represent the values of an enumeration type (§3.5.3).**

The values of enumeration types are represented as `ints`.

B.10 Qualifiers

- **What constitutes an access to an object that has volatile-qualified type (§3.5.3).**
  An access to an object that has volatile-qualified type is a 'read' from the memory location containing the object (if the object's value is required), or a 'write' to the memory location containing the object (if the object is assigned to). If the volatile object is an array, then the access will be only to the appropriate element of the array. If the volatile object is a structure and only a field of the structure is required, then the access will be only to the appropriate field. If the object is not an array element or structure field, then the object occupies a whole number of words, and all the words will be accessed. Otherwise, if the array element or structure field is shorter than a word, then only the appropriate bytes will be accessed.

  If the object is a bit-field, then in the case of read access, the entire word containing the bit-field will be read; and in the case of write access, the entire word containing the bit-field will be first read, and then written.

  Note that if the object is an array element or structure field of type `short` on a 32-bit transputer, or if the object is larger than two words, then the transputer block move instruction is used for the access. On some transputers, if a block move instruction is interrupted, when it resumes it may reread the same word of memory which was read immediately before the interrupt. This may cause problems with some peripheral devices.

B.11 Declarators

- **The maximum number of declarators that may modify an arithmetic, structure, or union type (§3.5.4).**
  There is no restriction upon the number of declarators that may modify an arithmetic, structure, or union type.

B.12 Statements

- **The maximum number of case values in a switch statement (§3.6.4.2).** There is no restriction upon the number of case values in a switch statement.

B.13 Preprocessing directives

- **Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Whether such a character constant may have a negative value (§3.8.1).**
  The value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Such a character constant may NOT have a negative value.

- **The method for locating includable source files (§3.8.2).**
  See section 11.3.1 in the accompanying user manual.

- **The support of quoted names for includable source files (§3.8.2).**
  See section 11.3.1 in the accompanying user manual.

- **The mapping of source file character sequences (§3.8.2).**
  See section 11.3.1. in the accompanying user manual.

- **The behaviour on each recognised `#pragma` directive (§3.8.6).**
  See section 11.3.11 in the accompanying user manual.

- **The definitions for `__DATE__` and `__TIME__` when respectively, the date and time of translation are not available (§3.8.8).**
  When date of translation is not available, `__DATE__` expands to:

  ```
  "Jan 1 1900"
  ```

  When time of translation is not available, `__TIME__` expands to:

  ```
  "00:00:00"
  ```

B.14 Library functions

- **The null pointer constant to which the macro `NULL` expands (§4.1.5)**
  `(void *)0`

- **The diagnostic printed by and the termination behaviour of the `assert` function (§4.2)**

  `*** assertion failed:` *condition*`, file` *file*`, line` *line*

  `assert` terminates by calling `abort`. The action of `abort` depends upon the use of the `set_abort_action` function. See the specification of `abort` in chapter 2.

- **The sets of characters tested for by the `isalnum`, `isalpha`, `iscntrl`, `islower`, `isprint` and `isupper` functions (§4.3.1)**

  `isalnum` : '0'-'9' 'A'-'Z' 'a'-'z'

  `isalpha` : 'A'-'Z' 'a'-'z'

  `iscntrl` : character codes 0-31 and 127

  `islower` : 'a'-'z'

  `isprint` : character codes 32-126

  `isupper` : 'A'-'Z'

- **The values returned by the mathematics functions on domain errors (§4.5.1)**
  All mathematics functions return the value `0.0` on domain errors.

- **Whether the mathematics functions set the integer expression `errno` to the value of the macro `ERANGE` on underflow errors. (§4.5.1)**
  The maths functions do set `errno` to `ERANGE` on underflow errors.

- **Whether a domain error occurs or zero is returned when the `fmod` function has a second argument of zero. (§4.5.6.4)**
  If the second argument to `fmod` is zero then a domain error occurs and the function returns zero.

- **The set of signals for the `signal` function (§4.7.1.1)**

  `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGINT`, `SIGSEGV`, `SIGTERM`, `SIGIO`, `SIGURG`, `SIGPIPE`, `SIGSYS`, `SIGALRM`, `SIGWINCH`, `SIGLOST`, `SIGUSR1`, `SIGUSR2`, `SIGUSR3`.

- **The semantics for each signal recognised by the `signal` function (§4.7.1.1)**

`SIGABRT`	Abnormal termination, such as initiated by the `abort` function.
`SIGFPE`	Erroneous arithmetic operation, such as zero divide or an operation resulting in overflow.
`SIGILL`	Detection of an invalid function image, such as an illegal instruction.
`SIGINT`	Receipt of an interactive attention signal.
`SIGSEGV`	Invalid access to storage.
`SIGTERM`	Termination request sent to the program.
`SIGIO`	Input/output possible.
`SIGURG`	Urgent condition on IO channel.
`SIGPIPE`	Write on pipe with no-one to read.
`SIGSYS`	Bad argument to system call.
`SIGALRM`	Alarm clock.
`SIGWINCH`	Window changed.
`SIGLOST`	Resource lost.
`SIGUSR1`	User-defined signal 1.
`SIGUSR2`	User-defined signal 2.
`SIGUSR3`	User-defined signal 3.

- **The default handling and the handling at program startup for each signal recognized by the `signal` function. (§4.7.1.1)**
The handling at program startup is identical to the default handling, which is as follows:

`SIGABRT`	The action of `SIGABRT` depends upon the `set_abort_action` function. See the specification of `abort` in chapter 2.
`SIGFPE`	No action.
`SIGILL`	No action.
`SIGINT`	No action.
`SIGSEGV`	No action.
`SIGTERM`	Terminate the program via a call of `exit` with the parameter `EXIT_FAILURE`.
`SIGIO`	No action.
`SIGURG`	No action.
`SIGPIPE`	No action.
`SIGSYS`	No action.

`SIGALRM`	No action.
`SIGWINCH`	No action.
`SIGLOST`	No action.
`SIGUSR1`	No action.
`SIGUSR2`	No action.
`SIGUSR3`	No action.

- **If the equivalent of `signal(sig, SIG_DFL);` is not executed prior to the call of a signal handler, the blocking of the signal that is performed (§4.7.1.1)**
  The equivalent of `signal(sig, SIG_DFL);` is executed prior to the call of a signal handler.

- **Whether the default handling is reset if the `SIGILL` signal is received by a handler specified to the signal function (§4.7.1.1)**
  The default handling is reset if the `SIGILL` signal is received.

- **Whether the last line of a text stream requires a terminating newline character. (§4.9.2)**
  The last line of a text stream does not require a terminating newline character.

- **Whether space characters that are written out to a text stream immediately before a newline character appear when read in. (§4.9.2)**
  Space characters written out to a text stream immediately before a newline character do appear when read in.

- **The number of null characters that may be appended to data written to a binary stream. (§4.9.2)**
  No null characters are appended to data written to a binary stream.

- **Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file. (§4.9.3)**
  The file position indicator of an append mode stream is initially positioned at the end of the file.

- **Whether a write on a text stream causes the associated file to be truncated beyond that point. (§4.9.3)**
  A write on a text stream will not cause the associated file to be truncated beyond that point.

- **The characteristics of file buffering. (§4.9.3)**
  When a stream is unbuffered characters appear from the source or destination as soon as possible.

  When a stream is line buffered characters are transmitted to and from the

host environment as a block when a newline character is encountered.

When a stream is fully buffered characters are transmitted to and from the host environment as a block when a buffer is filled.

In all buffering modes characters are transmitted when the buffer is full and when input is requested on an unbuffered or line buffered stream, or when the stream is explicitly flushed.

See also section 1.3.12.

- **Whether a zero length file actually exists (§4.9.3)**
  The library can support a zero length file if it is permitted on the host environment.

- **The rules for composing valid file names. (§4.9.3)**
  The rules for composing valid file names are the same as those found on the host system.

- **Whether the same file can be opened multiple times. (§4.9.3)**
  Although the system will allow a file to be opened multiple times the *icc* `stdio` library has no support for shared access to a single file and so unexpected results may occur if this is attempted.

- **The effect of the `remove` function on an open file. (§4.9.4.1)**
  The `remove` function will delete an open file only if this is permitted on the host system.

- **The effect if a file with the new name exists prior to the call to the `rename` function. (§4.9.4.2)**
  The rename will cause an existing file with the new name to be overwritten only if this is permitted on the host system.

- **The output for `%p` conversion in the `fprintf` function. (§4.9.6.1)**
  The output for the `%p` function is a hexadecimal number.

- **The input for the `%p` conversion in the `fscanf` function. (§4.9.6.2)**
  The input for the `%p` conversion is a hexadecimal number.

- **The interpretation of a - character that is neither the first nor the last character in the scanlist for `%[` conversion in the `fscanf` function. (§4.9.6.2)**
  A - character is treated in the same manner as all other characters no matter where it appears in the scan set.

- **The value to which the macro `errno` is set by the `fgetpos` or `ftell` function on failure. (§4.9.9.1, §4.9.9.4)**

errno is set to the value EFILPOS by the ftell or fgetpos function on failure.

- **The messages generated by the perror function. (§4.9.10.4)**

Value of errno	Message
0 (zero)	**No error (errno = 0)**
EDOM	**EDOM - function argument out of range**
ERANGE	**ERANGE - function result not representable**
ESIGNUM	**ESIGNUM - illegal signal number to signal()**
EIO	**EIO - error in low level server I/O**
EFILPOS	**EFILPOS - error in file positioning functions**
default	**Error code (errno) *errno* has no associated message**

- **The behaviour of the calloc, malloc, or realloc function if the size requested is zero. (§4.10.3)**
  If the size requested is zero in calloc or malloc then no action is taken and and the functions return NULL.

  If the size requested is zero in realloc and the pointer parameter is NULL then no action is taken and the function returns NULL. The case where size is zero and the pointer is not a NULL pointer is defined by the ANSI standard.

- **The behaviour of the abort function with regard to open and temporary files. (§4.10.4.1)**

  The abort function will cause termination without closing open files or removing temporary files. Note that the behaviour of abort may be altered by set_abort_action (see specification of the function in chapter 2) but whichever behaviour is selected, open files will not be closed, and temporary files will not be removed.

- **The status returned by the exit function if the value of the argument is other than zero, EXIT_SUCCESS, or EXIT_FAILURE. (§4.10.4.3)**
  The status returned by the exit function in this case is the numerical value of the argument.

- **The set of environment names and the method for altering the environment list used by the getenv function.(§4.10.4.4)**
  The set of environment names is defined by the host system.

  The method of altering the environment list on a given system is particular

to the server executing on that system. (Or, more accurately, particular to the compiler with which the server was compiled).

- **The contents and mode of execution of the string by the `system` function. (§4.10.4.5)**
  The string shall contain any of the commands which can be supported by the host operating system. Care should be taken so that no commands are issued which would cause the transputer to be booted, thereby overwriting the program which executed the system call.

  The mode of execution is defined by the host system.

- **The contents of the error message strings returned by the `strerror` function. (§4.11.6.2)**
  These are identical to the messages printed by the `perror` function. See above.

- **The local time zone and Daylight Saving Time. (§4.12.1)**
  The local time zone is defined by the host system. Daylight Saving Time information is unavailable.

- **The era for the `clock` function. (§4.12.2.1)**
  The era for the `clock` function extends from directly before the users main function is called until program termination.

B.15 Locale-specific behaviour

- **The content of the execution character set, in addition to the required members. (§2.2.1)**
  The execution character set comprises all 256 values 0 - 255. Values 0 - 127 represent the ASCII character set.

- **The direction of printing. (§2.2.2)**
  Printing is from left to right.

- **The decimal-point character. (§4.1.1)**
  The decimal point character is '`.`'.

- **The implementation defined aspects of character testing and case mapping functions (§4.3)**
  The only locale supported is "C" and so there are no implementation defined aspects of character testing or case mapping functions.

- **The collation sequence of the execution character set. (§4.11.4.4)**
  Only the C locale is supported and so the collation sequence of the execution character set is the same as as for plain ASCII.

- **The formats for time and date.** (§4.12.3.5)
  All the day and month names are in English.

date and time format:	`Thu Nov 9 15:42:39 1989`
date format:	`Thu Nov 9, 1989`
time format:	`15:42:39`

Index

inmos

Worldwide Headquarters

INMOS Limited
1000 Aztec West
Almondsbury
Bristol BS12 4SQ
UNITED KINGDOM
Telephone (0454) 616616
Fax (0454) 617910

Worldwide Business Centres

USA

INMOS Business Centre
Headquarters (USA)
SGS-THOMSON Microelectronics Inc.
2225 Executive Circle
PO Box 16000
Colorado Springs
Colorado 80935-6000
Telephone (719) 630 4000
Fax (719) 630 4325

SGS-THOMSON Microelectronics Inc.
Sales and Marketing Headquarters (USA)
1000 East Bell Road
Phoenix
Arizona 85022
Telephone (602) 867 6100
Fax (602) 867 6102

INMOS Business Centre
SGS-THOMSON Microelectronics Inc.
Lincoln North
55 Old Bedford Road
Lincoln
Massachusetts 01773
Telephone (617) 259 0300
Fax (617) 259 4420

INMOS Business Centre
SGS-THOMSON Microelectronics Inc.
9861 Broken Land Parkway
Suite 320
Columbia
Maryland 21045
Telephone (301) 995 6952
Fax (301) 290 7047

INMOS Business Centre
SGS-THOMSON Microelectronics Inc.
200 East Sandpointe
Suite 650
Santa Ana
California 92707
Telephone (714) 957 6018
Fax (714) 957 3281

INMOS Business Centre
SGS-THOMSON Microelectronics Inc.
2620 Augustine Drive
Suite 100
Santa Clara
California 95054
Telephone (408) 727 7771
Fax (408) 727 1458

INMOS Business Centre
SGS-THOMSON Microelectronics Inc.
1310 Electronics Drive
Carrollton
Texas 75006
Telephone (214) 466 8844
Fax (214) 466 7352

ASIA PACIFIC

Japan

INMOS Business Centre
SGS-THOMSON Microelectronics K.K.
Nisseki Takanawa Building, 4th Floor
18–10 Takanawa 2-chome
Minato-ku
Tokyo 108
Telephone (03) 280 4125
Fax (03) 280 4131

Singapore

INMOS Business Centre
SGS-THOMSON Microelectronics Pte Ltd.
28 Ang Mo Kio Industrial Park 2
Singapore 2056
Telephone (65) 482 14 11
Fax (65) 482 02 40

EUROPE

United Kingdom

INMOS Business Centre
SGS-THOMSON Microelectronics Ltd.
Planar House
Parkway Globe Park
Marlow
Bucks SL7 1YL
Telephone (0628) 890 800
Fax (0628) 890 391

France

INMOS Business Centre
SGS-THOMSON Microelectronics SA
7 Avenue Gallieni
BP 93
94253 Gentilly Cedex
Telephone (1) 47 40 75 75
FAX (1) 47 40 79 27

West Germany

INMOS Business Centre
SGS-THOMSON Microelectronics GmbH
Bretonischer Ring 4
8011 Grasbrunn
Telephone (089) 46 00 60
Fax (089) 46 00 61 40

Italy

INMOS Business Centre
SGS-THOMSON Microelectronics SpA
V.le Milanofiori
Strada 4
Palazzo A/4/A
20090 Assago (MI)
Telephone (2) 89213 1
Fax (2) 8250449